Declarative, Parallel Programming For End-User Development

Alexander Asp Bock

Advisor: Peter Sestoft
Submitted: March 26, 2019

IT UNIVERSITY OF COPENHAGEN

Dedicated to my mother and father for their unending love and support.

Abstract

Spreadsheets are popular tools in many areas such as science, finance and engineering for large, complex models that inform critical decisions. Spreadsheet end-users are usually domain experts but may not be trained IT professionals.

Shared-memory multicore processors have become commonplace commodities, but how can spreadsheet end-users leverage this powerful hardware to accelerate spreadsheet computation? They may need to collaborate with an IT department or consult professionals.

Existing solutions usually require some level of user interaction or the availability of a compute cluster. This thesis investigates the development of tools and algorithms that automatically accelerate spreadsheet recalculation in a completely transparent manner, targeting the shared-memory multicore processors of today's computers. The declarative and functional nature of spreadsheets makes them a prime candidate for automatic parallelism.

We explore two approaches to automatic parallelism. The first *dynamically* attempts to discover *local* parallelism on-the-fly during cell evaluation. The second approach *statically* analyses the spreadsheet and *globally* partitions cells into load-balanced groups that can be executed in parallel on shared-memory multicore processors.

The work is implemented in Funcalc, a research spreadsheet application featuring higher-order functions. They can be defined directly inside the spreadsheet using the formula language end-users are already familiar with. We believe this additional expressive power combined with automatic tools for levering the underlying hardware is a powerful framework for end-user development.

Resumé

Regneark er populære værktøjer der bruges i mange fagområder såsom videnskab, finans og ingeniørarbejde til håndtering af store, komplekse modeller, der bruges til at foretage kritiske beslutninger. Slutbrugere af regneark er typisk domæneeksperter men ikke nødvendigvis professionelle IT-folk.

Flerkerne-processorer er blevet almindelige i nutidens computere, men hvordan kan slutbrugere udnytte denne kraftige hardware til at accelerere regnearksberegninger? De kan være nødt til at samarbejde med en IT afdeling eller søge rådgivning fra eksperter.

Eksisterende løsninger behøver typisk interaktion fra brugeren eller en tilgængelig klyngecomputer. Denne afhandling udforsker udvikling af værktøjer og algoritmer, der automatisk kan accelerere regnearksberegninger fuldstændig transparent for brugeren og målrettet nutidens flerkerne-processorer. Regnearks deklarative og funktionelle aspekter gør dem til en oplagt kandidat til automatisk parallelisme.

Vi udforsker to tilgange til automatisk parallelisme. Den første tilgang forsøger dynamisk at finde lokal parallelisme, mens cellerne i regnearket beregnes. Vores anden tilgang udfører en statisk analyse af regnearket og partitionerer globalt cellerne i grupper, der ligeligt fordeler det totale arbejde i regnearket. Disse grupper kan derefter eksekveres i parallel på flerkerne-processorer.

Afhandlingens arbejde er udført i Funcalc, et regnearksprogram til forskning, hvor brugere kan definere højere ordens funktioner direkte i regnearkets celler i det formelsprog de allerede er bekendte med. Vi tror denne forbedrede udtrykskraft kombineret med automatiske værktøjer til udnyttelse af den underliggende hardware vil understøtte og væsentligt forbedre slutbrugerudvikling i regneark.

Résumé

Les logiciels tableurs sont des outils populaires dans beaucoup de domaines comme la science, la finance et l'ingénierie permettant la prise de décisions critiques par le biais de l'usage de modèles complexes de vaste taille. Les utilisateurs finaux sont typiquement des experts dans leur domaine mais ne sont pas forcément des professionnels dans le domaine informatique.

Les processeurs «multi-cœurs» à mémoire partagée sont maintenant d'usage courant, mais on peut se poser la question comment les utilisateurs finaux peuvent-ils tirer parti de ce puissant matériel pour accélérer les calculs du tableur ? Ils peuvent avoir besoin de collaborer avec un service informatique ou de consulter des professionnels. Les solutions existantes nécessitent généralement un certain niveau d'interaction de l'utilisateur ou la disponibilité d'un cluster de calcul. Cette thèse étudie le développement d'outils et d'algorithmes qui permettent d'accélérer le recalcul des feuilles de calcul de manière totalement transparente basés sur des processeurs multi-cœurs à mémoire partagée utilisés dans les ordinateurs d'aujourd'hui. Le caractère déclaratif et fonctionnel des tableurs les rend candidats idéals pour le parallélisme automatique. Nous explorons deux approches du parallélisme automatique. La première tente de façon dynamique de découvrir le parallélisme local à la volée pendant le calcul des feuilles. La seconde approche analyse de façon statique les feuilles de calcul et partitionne globalement les cellules en groupes équilibrés qui peuvent être exécutés en parallèle par des processeurs multi-cœurs à mémoire partagée.

Notre travail est réalisé dans Funcalc, un tableur de recherche permettant de définir des fonctions d'ordre supérieur directement dans les feuilles de calcul dans le langage de formule déjà connu par les utilisateurs finaux. Nous sommes d'avis que cette puissance expressive supplémentaire combinée avec des outils automatiques exploitant le matériel informatique disponible va supporter et améliorer considérablement les possibilités pour le développement des utilisateurs finaux des tableurs.

Acknowledgements

There have been many trials and tribulations during my time as a PhD student. A couple of people, listed in no particular order, have made this journey much more pleasant and been provided some of its highlights.

First and foremost, I would like thank my supervisor Peter Sestoft for the opportunity to become a PhD student as well as for guidance, advice and support. It has been an extraordinary personal experience and allowed me to delve deeper into various fascinating subjects.

Thanks to my colleague Florian Biermann for his advice, help and invaluable discussions.

Thanks to Bent Thomsen, Thomas Bøgholm and Lone Leth Thomsen of Aalborg University for a fruitful and educational collaboration.

Thanks to Claus Brabrand for inviting me to the Software Quality Research group, his advice, assistance and willingness to lend a helping hand. I, and the rest of the computer science department I suspect, are grateful for his donation of a good coffee maker to the department kitchen.

I would like to thank Gul Agha from the University of Illinois, Urbana-Champaign for hosting me as a visiting scholar in 2017. I would also like to thank Karl Palmskog for his support during my stay at UIUC and showing me the best restaurants.

Likewise, thanks to Claudio Russo for hosting me at Microsoft Research, Cambridge in the same year and providing me with one of the definitive highlights of my PhD studies.

Thanks to our two student programmers, Holger Stadel Borum and Malthe Kirkbo, for their work on Funcalc.

Thanks to the Independent Research Fund Denmark for funding this work.

I am grateful for my girlfriend, Malene, without whom this long jour-

ney would have been much less enjoyable. Her endurance and support during rough patches were indispensable.

I would like to thank my brother Andreas for his advice when I was considering the position as a PhD student and lending a helping hand. Thanks also to my Mum and Dad for all their support!

Last but not least, a huge thanks to the following people for proof-reading articles and the thesis itself: Peter Sestoft, Claus Brabrand, Kenneth Ry Ulrik, Holger Stadel Borum, Andreas Bock and Karl Palmskog.

Summary of Papers

1. "**Puncalc: Task-Based Parallelism and Speculative Reevaluation in Spreadsheets**" [1, 2]

   Alexander Asp Bock and Florian Biermann

   Abstract Spreadsheets are commonly declarative, first-order functional pro- grams and are used as organizational tools, for end-user development, and for educational purposes. Spreadsheet end-users are usually domain experts who use spreadsheets as their main computational model, but are seldom trained IT professionals who can leverage today's abundant multicore processors for spreadsheet computation. In this paper, we present an algorithm for auto- matic, parallel evaluation of spreadsheets targeting shared-memory multicore architectures, that lets end-users transparently make use of their multicore processors. We evaluate our algorithm on a set of synthetic and real-world spreadsheets, and obtain up to 16 times speedup on 48 cores.

2. "**Static Partitioning of Spreadsheets for Parallel Execution**" [3]

   Alexander Asp Bock

   Abstract Spreadsheets are popular tools for end-user development and complex modelling but can suffer from poor performance. While end- users are usually domain experts they are seldom IT professionals that can leverage today's abundant multicore architectures to offset such poor performance. We present an iterative, greedy algorithm for automati- cally partitioning spreadsheets into load-balanced, acyclic groups of cells that can be scheduled to run on shared-memory multicore processors. A big-step cost semantics for the spreadsheet formula language is used to estimate work and guide partitioning. The algorithm does not require end-users

to modify the spreadsheet in any way. We implement three extensions to the algorithm for further accelerating computation; two of which recognise common cell structures known as cell arrays that natu- rally express a degree of parallelism. To the best of our knowledge, no such automatic algorithm has previously been proposed for partitioning spreadsheets. We report a maximum 24-fold speed-up on 48 logical cores.

3. "**A Parallel Spreadsheet Interpreter With Cycle Detection**" (submitted to ECOOP 2019)

   Alexander Asp Bock

   Abstract It was estimated that there would be 72 million spreadsheet users monthly in 2017 some of which build complex financial, scientific and mathematical models. In the age of multicore computing and ever-increasing amounts of data, how can end-users, who may not be IT professionals, access this powerful hardware to accelerate spreadsheet computation?

   Existing solutions are usually not fully automatic and often require end-users to modify the spreadsheet in some way to facilitate parallel execution. Ideally, end-users should have a tool at their disposal to transparently exploit the underlying hardware and automatically discover available parallelism in the spreadsheet. This paper presents such an algorithm which also features parallel cycle detection. We implement the algorithm in the Funcalc research spreadsheet application that introduces higher-order, user-defined functions to the spreadsheet paradigm. Along with automatic parallel execution, this is a powerful and expressive platform for end-user development.

   Our results show a maximum 5.55–21.34x speed-up on a set of benchmark spreadsheets, a 4.15–4.89x speed-up on a set of synthetic spreadsheets and overall improvements over previous work.

Summary of Technical Reports

1. "**A Literature Review of Spreadsheet Technology**" [4]

   Alexander Asp Bock

   It was estimated that there would be over 55 million end-user programmers in 2012 [5] in many different fields such as engineering, insurance and banking, and the numbers are not expected to have dwindled since. Consequently, technological advancements of spreadsheets is of great interest to a wide number of people from different backgrounds. This literature review presents an overview of research on spreadsheet technology, its challenges and its solutions. We also attempt to identify why software developers generally frown upon spreadsheets and how spreadsheet research can help alter this view.

2. "**Concrete and Abstract Cost Semantics for Spreadsheets**" [6]

   Alexander Asp Bock, Thomas Bøgholm, Peter Sestoft, Bent Thomsen and Lone Leth Thomsen

   We give a simple but precise operational semantics for the evaluation of extended spreadsheet formulas, with array formulas, sheet-defined functions and closures, as found in the Funcalc spreadsheet platform.

   We build on this to give a simple cost semantics for evaluation of a spreadsheet formula and for full and minimal recalculation of a spreadsheet.

   Following the ideas presented by Schmidt we provide a big step trace-based abstract interpretation for the cost semantics.

   We then present a set of functions which can be used to calculate the cost of executing an evaluation of a spreadsheet expression

following Gomez et al., inspired by Rosendahl. These functions are related to the above operational semantics, cost semantics and abstract interpretation.

The above semantic presentations all form the formal foundations for various cost calculations implemented in the Funcalc spreadsheet platform.

3. "**A Comparison Between SISAL 1.2 and Funcalc**" [7]

   Alexander Asp Bock

   We translated 16 programs, written in the Streams and Iteration in a Single Assignment Language (SISAL) language, of varying complexity to Funcalc using higher-order sheet-defined functions to demonstrate Funcalc's ability to express a variety of functional programs. The differences between SISAL's and Funcalc's type system and syntax are also discussed with directions for future work to further increase Funcalc's expressive capabilities.

Table of Contents

Chapter 1

Introduction

Spreadsheets continue to remain popular almost four decades after the invention of Dan Bricklin and Bob Frankston's VisiCalc [8]. Despite their success and mainstream appeal they have also been criticised [9–11] for their lack of the rigid programming doctrines expected by "real" programmers. In a 1992 paper, aptly titled "*Real Programmers Don't Use Spreadsheets*", Casimir writes:

> "... therefore, I suggest that spreadsheets are intrinsically uninteresting... The purpose of this paper is to list the few things that should be known about spreadsheets and to warn would-be researchers for the dullness of the subject." ([10])

In a more recent paper from 2004, Wile writes:

> "One must realize that spreadsheets are a success not because they allow programming, but rather in spite of the fact that they allow it!" ([11, p. 280])

Regardless, spreadsheets continue to remain popular tools for end-user development today in areas such as finance and science for several reasons in part being an amalgamation of a first-order functional language and a visual, data-driven tool. For example, users can modify cells in the spreadsheet to get immediate visual feedback on the effects of the modification. Spreadsheets support convenient bulk operations on cells such as summation and powerful copy-paste facilities. As a functional language, they are typically side-effect free so users do in general not have to worry about state. Scaffidi [12] estimated in 2017

that there would be 72 million professional spreadsheet users in that same year. When viewed as functional languages [13, 14], that makes spreadsheets one of the most widely used forms of functional programming! This alone makes spreadsheets an interesting area of research with the chance of new ideas benefiting millions of people.

The dichotomy of opinions on spreadsheets might stem partly from the lack of academic involvement in their development [15]. Consequently, spreadsheets have not been subject to the same rigorous principles enjoyed by "real" programmers of non-spreadsheet programming languages. Interestingly, the research community's seeming disinterest in spreadsheets has been discussed at least as far back as 1985 by Mani Chandy [15] in his invited address at the Principles of Distributed Computing (PODC) conference.

> "Software packages written using spreadsheets are now sold, and in some cases, these packages display little concern for the qualities espoused by academic computing sciences: correctness, simplicity, elegance, understandability and maintainability." ([15])

The cumulative effect of these shortcomings likely contributes further to the negative view of spreadsheets. Fortunately, the tide appears to have shifted since 1985 and there is now much more active research on spreadsheets as we survey in chapter 3. In this thesis, *we follow suit and explore how different approaches to one such quality of computer science, namely* automatic *or* implicit *parallelism, can be integrated into the spreadsheet paradigm to accelerate spreadsheet computation*. The automation is especially important as spreadsheet end-users are usually non-programmers who have limited or no training in IT, but are domain experts. Therefore, they should focus on problem solving not parallelisation.

When on the subject of parallelism, one must inevitably mention that the "free lunch is over" [16] as stated by Herb Sutter in a 2005 article on the coming trends in concurrency and computer architecture. The gist was that due to physical limitations in processor chips, we cannot simply buy a new, faster processor to accelerate sequential computation. In response, hardware has instead evolved to utilise multiple shared-memory processors to increase performance, leading to a recent rejuvenated age of parallel computing [17]. The present work is partly spurred

by this new age and today's wide availability of shared-memory multicore systems in everything from personal laptops to mobile phones.

The idea of parallel spreadsheets is not new. In fact, Mani Chandy also addressed this idea in his talk [15], suggesting a model for distributed spreadsheets wherein non-programmers could specify their problems, shielded from the details of the underlying implementation. The spreadsheet could then be compiled to a distributed programming language and executed on parallel hardware.

His suggestion and our work are both motivated by several attractive properties of the spreadsheet languages that facilitate parallel execution. We have already mentioned one of these properties: side-effect freedom which means that multiple threads can in general operate on distinct cells without having to worry about different execution interleavings producing different results. Another property is cell immutability. The formula language seldom provides ways in which to modify cells directly. Instead one must transform existing data so multiple threads do not need to worry about concurrent modifications to a cell. Both side-effect freedom and immutability originate in functional programming languages and have long been heralded as some of the major strengths of such languages [18, 19].

Spreadsheets are also declarative languages. End-users focus on specifying *what* needs to be computed, letting the implementation care about *how* it gets computed, as Mani Chandy also drew attention to [15]. Spreadsheets are also unique as functional languages in that they provide a grid of cells and an input-driven dataflow model of computation: when a cell is modified, the new information automatically flows forward to cells that directly or transitively depend on the modified cell, with immediate visual feedback. The dataflow system is constrained by cell dependencies, another property of spreadsheets that makes them attractive for parallelisation as independent cells can be evaluated in parallel.

The attractive features of spreadsheets combined with their popularity can serve as a medium for parallel computing to reach a wide audience of non-programmers [15]. This is the goal of the Popular Parallel Programming (P3) project [20] which this dissertation is part of.

The need for parallel recalculation of spreadsheets may not be immediately obvious but they are used to develop large, complex models in finance and science. One example is a Monte Carlo simulation [21] which took the authors 431 working hours to drastically improve the

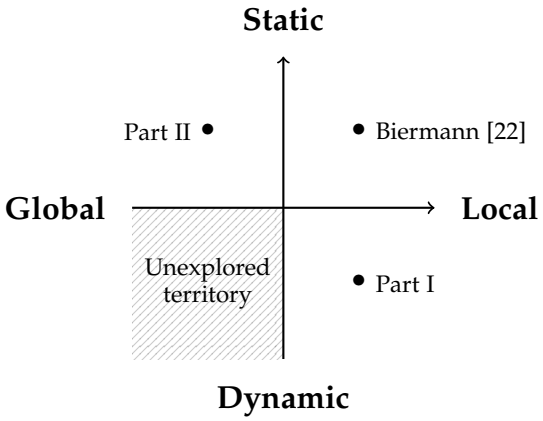


Figure 1.1 – A two-dimensional spectrum of parallelisation strategies in spreadsheets. Parts I and II explore opposing quadrants of this spectrum.

performance of the spreadsheet. Instead of relying on such expert help, we instead wish to develop tools and algorithms that can automatically find opportunities for parallelism in the spreadsheet with little or no required interaction from the user. Other work has attempted to achieve a similar goal but we defer a discussion until chapter 3.

Figure 1.1 shows a two-dimensional spectrum of approaches to parallelism and the combinations explored in this thesis and other work. The vertical axis spans static compile-time and dynamic runtime and the horizontal axis spans global and local. In a static approach, the program is statically analysed to uncover opportunities for parallelism. In a spreadsheet, this could be done at load-time when it is initially loaded or during certain types of cell computation that evaluates all cells in the spreadsheet. In contrast, dynamic approaches attempt to uncover parallelism as the program executes. This could be done during cell evaluation in a spreadsheet. The global and local spectrum indicate the level of granularity at which parallelism is extracted. In a program context, extracting parallelism from the entire program might correspond to a global approach whereas extracting parallelism from individual statements might correspond to a local approach. In a spreadsheet, we could either extract parallelism from the spreadsheet as a whole or instead locally at the level of cells or even at the level of individual formula expressions.

This thesis has focused on two opposite quadrants of the spectrum: a

local, dynamic approach discussed in part I and a global, static approach in part II. The work of Biermann et al. [22] has focused on a static, local approach to automatic parallelism while the combination of global and dynamic approaches remains unexplored in the context of spreadsheets.

We firmly believe that reading this dissertation will help convince readers that spreadsheets are much more than a simple, dull modelling tool which has intricate inner workings and design trade-offs that affect everything from performance to end-user development. Contrary to the quoted bleak words of Casimir [10], there are many exciting venues for research to contribute to the future development of spreadsheets to the benefit of millions of users.

1.1 Thesis Outline and Contributions

The outline for the remainder of the thesis is as follows. In chapter 2, we introduce some important concepts and terminology at the heart of spreadsheets which will bring readers up to speed for the next two chapters. Chapter 3 gives some overall background, specifically the concept of *dataflow*, its history and how it relates to spreadsheets. This is followed by a brief history of spreadsheets from their inception to contemporary systems and a survey of spreadsheet research. In chapter 4, we describe Funcalc, the spreadsheet application used to implement our work and how it differs from other spreadsheet applications.

The rest of the dissertation is split into three primary parts. The first two describe our different approaches to automatic parallelism given by the two-dimensional spectrum in figure 1.1. The last part summarises our findings and gives directions for future work.

Part I

In the first major part of the thesis, we introduce our local, dynamic parallel algorithms. In chapter 5, we first discuss the changes necessary to make Funcalc thread-safe, then present a task-based parallel cell interpreter for spreadsheets along with a method for detecting cyclic dependencies in parallel. A thorough investigation of the task-based interpreter's performance characteristics follows in chapter 6 and new insights lead to an improved parallel interpreter in chapter 7. As an academic curiosity, we develop a different parallel cycle detection method

for the new interpreter inspired by a distributed cycle detection algorithm that propagates information about reachability among threads.

Part II

The second major part of the thesis presents a cost model in chapter 8 that models the cost associated with synchronisation between parallel processes and a concrete big-step cost semantics for estimating the cost of evaluating individual cells. The cost model is first and foremost used to guide the partitioning algorithm, presented in the following chapter, in balancing the trade-off between synchronisation and parallelism. We also discuss other applications of the natural semantics besides partitioning.

The static partitioning algorithm is presented next in chapter 9. It globally analyses the spreadsheet using the cost model then partitions cells into groups that can efficiently be executed in parallel on shared-memory multicore systems. Three different extensions for exploiting additional parallelism are proposed and implemented. The extensions rely on certain groups of cell structures commonly found in spreadsheets.

Part III

The final part summaries and discusses the principal findings in our work as well as multiple interesting directions for future work.

Chapter 2

Spreadsheet Concepts

We introduce some core spreadsheet concepts and terminology in this chapter for readers unfamiliar with the subject. An understanding of these concepts will prove useful when discussing the background of the thesis and spreadsheet research in chapter 3 as well as the Funcalc spreadsheet application in chapter 4. Readers interested in learning more about these subjects are encouraged to read [23].

2.1 Workbook and Sheets

Figure 2.1 shows the typical hierarchy in a spreadsheet application. A workbook is usually the term used for the entire spreadsheet file and each workbook may contain multiple sheets usually organised into tabs. Each sheet in turn contains a two-dimensional grid of cells arranged into rows and columns that are uniquely addressable. Different cell addressing formats exist which we discuss in section 2.3. Each cell can contain different things which we discuss in the next section.

2.2 Cells and Formulas

A cell can usually contain either a constant, such as a number, a string, or an error value (e.g. `#NA` for "not available" or `#DIV/0!` for division by zero); or a formula expression denoted by a leading equals character (e.g. `=1+2`). Each cell has a unique address denoted by its column and row position with columns starting at A and rows at 1 in the A1 reference format. Formulas can refer to other cells using cell addresses

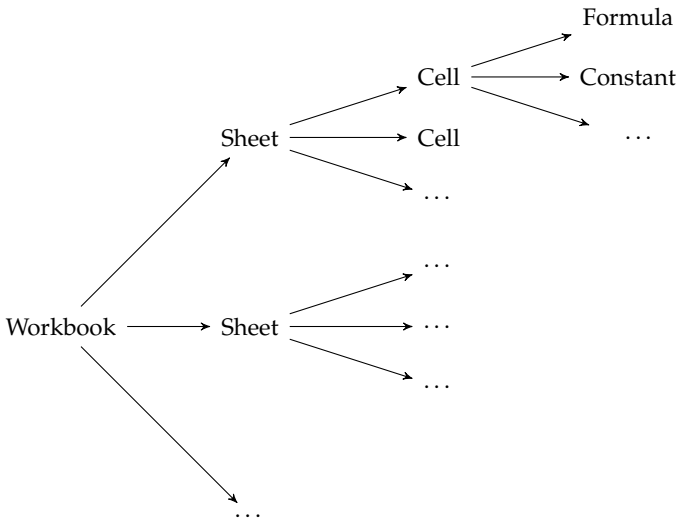


Figure 2.1 – The typical hierarchy of a spreadsheet. A workbook can contain multiple sheets that in turn can contain multiple, different cell types.

	A	B	C
1	10	20	=A1*A2
2	30	40	=$A2
3			=A1:B2

(a)

	A	B	C
1	10	20	=RC[-2]*R[+1]C[-2]
2	30	40	=R[+0]C1
3			=R[-2]C[-2]:R[-1]C[-1]

(b)

Figure 2.2 – An example spreadsheet in A1 and R1C1 reference formats. The latter format more clearly shows relative references.

(e.g. `=B2`) or refer to an area of cells using the addresses of any two opposing corner cells separated by a colon. For example, cell C1 in figure 2.2a refers to cells A1 and A2 while cell C3 refers to the 2×2 cell area spanned by A1 and B2.

2.3 Cell References

A cell reference can be relative or absolute in each dimension independently. In the A1 reference format, a dollar sign denotes an absolute row or column reference while its absence denotes a relative row or column reference. In cell C1 in figure 2.2a the cell references in the `=A1*A2` are thus both row-relative and column-relative. The reference `=$A2` in C2 refers absolutely to column A but relatively to row 2. A relative reference refers to other cells relative to its own position, so the referenced cell depends on the position of the referring cell. This is important when copy-pasting cells and why this operation is a powerful tool. Absolute references do not change when copied and always refer to the same row or column. Copying cell C2 to cell C3 would change the cell reference to `$A3`. It would remain unchanged if copied to D2 since the column reference is absolute and the cell remains in the same row as before.

Another reference format is the R1C1 format which more clearly expresses relative references. Note that the row now comes first followed by the column. Relative references are denoted by a bracketed offset whereas absolute references have an absolute row or column index and no brackets. Therefore, the cell reference `=R[-1]C2` would refer to a cell in the row above it in column 2. If neither a bracketed offset or a number is present it corresponds to a zero offset, thus `=RC2` equates to `=R[+0]C2`. We use an explicit zero offset in the remainder of the thesis to avoid confusion. To highlight the differences between the formats, the spreadsheet in figure 2.2a is shown in figure 2.2b in R1C1 format. The formula expression in cell C1 references two cells. The first references the cell in the same row and two columns to the left (cell A1), and the second references the cell one row below and two columns to the left (cell A2).

We also briefly mention the C0R0 reference format which uses zero-based offsets and switches the order of the row and column index. We will not be using this format in this thesis.

2.4 Cell Arrays

A cell array [24, 25] is a contiguous rectangular cell area containing formulas that share the same formula expression and thus the same computational semantics [24]. They are also known as copy-equivalent

	A	B
1	1	=A1*2
2	2	=A2*2
3	3	=A3*2

(a) Cell array in the A1 reference format in column B.

	A	B
1	1	=R[+0]C[-1]*2
2	2	=R[+0]C[-1]*2
3	3	=R[+0]C[-1]*2

(b) The same cell array in the R1C1 reference format.

	A	B
1	1	=R[+1]C[+0]*2
2	2	=R[+1]C[+0]*2
3	3	=R[+1]C[+0]*2

(c) A transitive cell array.

	A	B
1	1	=R[+0]C[-1]*2
2	2	=R[+0]C[-1]*2
3	3	=R[+0]C[-1]*2

(d) An intransitive cell array.

Figure 2.3 – Cell arrays in the A1 and R1C1 reference formats and examples of transitive and intransitive cell arrays.

formulas [26] or cp-similar cells [27] because they tend to arise from copy-paste operations. A 3×1 cell array is shown in column B in A1 reference in figure 2.3a format and in figure 2.3b in R1C1 format. The R1C1 format makes it clear that the formulas in the cell array share a common expression.

Cell arrays are common and useful in spreadsheets because they describe bulk operations on collections of cells similar to e.g. `map` and `reduce` on arrays in conventional functional programming languages. For example, the cell array in figure 2.3a effectively describes a `map` operation on column `A` that multiplies each cell by two. This demonstrates why copy-pasting cells in spreadsheets combined with relative references lead to bulk operations on cells. Dou et al. [24] found that 69% (7416 out of 10 754) of spreadsheets containing formulas from the EUSES [28] and Enron [29] spreadsheet corpora also contained cell arrays, and that they contained on average 80 cell arrays each.

A cell array can be classified as either *transitive* (figure 2.3c) or *intransitive* (figure 2.3d) [22]. If a cell array only contains formulas that do not reference the cell array itself, we say that it is intransitive, otherwise it is transitive. The need for this distinction will become clear later in chapter 9 when we describe the static partitioning algorithm and two of its extensions.

	A	B
1	11	12
2	21	22
3	=TRANSPOSE(A1:B2)	
4		

(a) Transposition of cell area A1:B2 in cells A3:B4.

	A	B
1	11	12
2	21	22
3	11	21
4	12	22

(b) Result of transposition.

	A	B
1	11	12
2	21	22
3	{=TRANSPOSE(A1:B2)}	{=TRANSPOSE(A1:B2)}
4	{=TRANSPOSE(A1:B2)}	{=TRANSPOSE(A1:B2)}

(c) Formula view of transposition.

Figure 2.4 – Transposition of a cell area. The area selected by the user is highlighted with a red border.

2.5 Array Formulas

When a user selects a cell area and enters a formula that returns an array, the elements of the array are distributed across the selected area. The cells in the area share the same singular formula expression but each cell refers only to one element of the array. Figure 2.4a shows how array formulas can be used to transpose the 2×2 cell area spanned by cells A1 and B2 using the `TRANSPOSE` function. The user initially selects the area A3:B4 (highlighted in red) then enters the array formula that calls the `TRANSPOSE` function with A1:B2 as argument and finally presses some key combination, depending on the spreadsheet application. The result of the transposition is shown in figure 2.4b. Figure 2.4c shows how the elements of an array formula are usually presented to the user where each cell contains the source expression enclosed in curly brackets.

The user may not select an area that fits the size of the resulting array. In Funcalc, selecting a smaller area simply truncates the results and selecting too large an area fills the cells that are out of bounds with `#NA` error values.

2.6 The Support and Dependency Graphs

The cell references in a formula are its dependencies on other cells. Therefore, cell references collectively establish a cell *dependency graph*. Its inverse is called the *support graph* and captures cell support. It is analogous to a dataflow graph [30] where the nodes are cells and data flow forward along the edges from dependencies to supported cells. We also refer to the supported cells of a cell as its *support set*. Cell C1 in figure 2.2a depends on A1 and A2 while both A1 and A2 support C1. For the remainder of this thesis, we use solid lines ($\longrightarrow$) to denote support graph edges and dashed lines ($\dashrightarrow$) to denote dependency graph edges as shown in figure 2.5b.

2.7 Recalculation

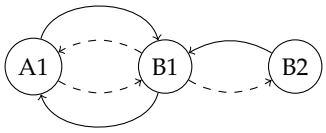
The purpose of recalculation is to bring the spreadsheet to a *consistent state* [23, sec. 1.8.3]. We return to the exact definition of consistent state in section 2.9 but suffice to say that it must find values for all cells that agree with their respective formulas. A cornerstone of spreadsheets is that this process happens automatically in response to user modifications. If a cell is modified, all cells that transitively depend on its value are automatically updated to reflect the change, with immediate visual feedback on the effect of the modification.

There are two major types of recalculation. *Full recalculation* unconditionally reevaluates all formula cells. *Minimal recalculation* reevaluates only the supported cells reachable from those cells that were modified by the user, i.e. the transitive closure of the support graph, starting from the modified cells as just explained. In figure 2.2a, whenever a user edits the value in A1, cells C1 and C3 must be updated to reflect the change but C2 does not. Constants involve no work and do not need to be recalculated.

In actuality, a minimal recalculation may evaluate more than just the cells modified by the user. Certain built-in functions are *volatile* which means that cells calling them must be recalculated regardless of whether they are changed or reachable from a modified cell. For example, most spreadsheets provide the `RAND` and `NOW` functions. The former usually produces a random number in the interval $[0, 1[$. If it were not volatile, it would produce a single random value which would immediately be-

	A	B
1	=B1+1	=IF(RAND()<0.5, A1, B2)
2		=NOW()

(a)



(b)

Figure 2.5 – (a) A small spreadsheet with a static cycle between A1 and B1 **(b)** Its corresponding cyclic graph with support edges and dependencies.

come stale after the first recalculation and we would lose the desired non-determinism of the function. Comparably, the `NOW` function returns the elapsed time since some epoch, and would produce a stale value if not recomputed each time. The user would manually have to update each cell to generate new values. Therefore, a minimal recalculation includes both the cells modified by the user and all volatile cells, collectively known as the *recalculation roots*. Of course, a full recalculation also evaluates these cells since it recalculates all cells in the spreadsheet.

Some spreadsheet applications, like Excel, also feature *manual* recalculation where the user has full control. As its name implies, recalculation only happens when requested by the user. In this mode, the cell values are not guaranteed to be consistent except after a normal recalculation.

As an aside, the first part of the thesis focuses on parallelisation of minimal recalculation since this involves an unknown amount of parallelism that is discovered dynamically. The second part focuses on parallelisation of full recalculation as it considers parallelism found at the global level of the spreadsheet.

2.7.1 Static and Dynamic Cycles

The dependency and support graphs can be cyclic. This occurs when two cells directly or transitively refer to one another. Cycles usually indicate an error since it may be impossible to find consistent values for cells without resolving the cycle, thus obstructing computation. However, they may sometimes be intentional. Excel allows for iterative, user-controlled recalculation of cyclic spreadsheets to support converging computations that repeat until a specific condition is met.

Static cycles describe *potential* cycles that can occur via the cell ref-

	A	B
1	=1	=10
2	=INDIRECT("B"&A1)	

Figure 2.6 – Example of using `INDIRECT` without statically referring to a cell in the formula. The cell ultimately referenced by A2 dynamically depends on the value of A1.

erences explicitly mentioned in a formula expression. Dynamic cycles describe *actual* cycles that occur when cells are being evaluated. How can a cycle be potential? To illustrate, consider the expression in cell B1 of figure 2.5a where taking one branch of the `IF` causes a cycle whereas taking the other does not. There is thus a static cycle between cells A1 and B1 that may result in a dynamic cycle at evaluation time based on which branch is ultimately taken. In a recalculation algorithm, we wish to detect dynamic cycles that occur during evaluation.

2.7.2 Dynamic Indexing Functions

Some functions like `INDIRECT`, that is found in both Excel and LibreOffice Calc, can dynamically refer to other cells and can thus have *dynamic dependencies* that are not known until evaluation time. `INDIRECT` accepts a cell reference, a cell area or a string as an argument. A string argument such as `"B1"` is interpreted as a cell reference. For example, to evaluate the formula in cell A2 of figure 2.6, the string `"B"` and the value of cell A1 are concatenated with the string concatenation operator &. The resulting string is interpreted as a cell reference which will refer to some cell in column B, but exactly which cell will depend on the value of cell A1. In this case, the cell will refer to B1 even though its cell address is never statically referenced by A2.

Excel handles dynamic dependencies by marking cells containing such calls as volatile [31]. This sacrifices minimality of a minimal recalculation in some cases since it may not be necessary to update a dynamic dependency between recalculations. However, it elegantly solves the problem of tracking dynamic dependencies or resorting to mutate the dependency graph. For example, if we modified cell B2 in figure 2.6, we would not need to update A2. Alternatively, a static analysis could exclude some uses of `INDIRECT` that would not change between recalculations.

e	::=	`n`	number constant
	|	`ca`	cell reference, e.g. B2 or G$6
	|	`IF`(e_1,e_2,e_3)	conditional expression
	|	`RAND`$()$	volatile function
	|	`F`$(e_1,\ldots,e_n)$	call to built-in function, e.g. `=SUM(A1, B2)`
	|	$\texttt{ca}_1 : \texttt{ca}_2$	cell area reference, e.g. A1:B2 or G$6:C1
	|	$ae[i,j]$	array formula component
ae	::=	e	array expression

Figure 2.7 – Syntax for a small spreadsheet formula language with array formulas [6].

2.8 A Formal Spreadsheet Language

We now present the syntax and semantics of a small spreadsheet language originally introduced by Sestoft [23] and extended in a technical report [6]. It partially describes the formula language of Funcalc. Based on the semantics, we define the consistency requirements for a recalculation in the next section and use them in discussion of the recalculation algorithms of this thesis. In chapter 8, we extend the semantics to include costs and several features of Funcalc such as first-class function values, user-defined functions and intrinsic functions.

Figure 2.7 shows the grammar for our small spreadsheet language. An expression e can be either a numerical constant; a cell reference; an `IF` expression where the result of evaluating e_1 determines whether the expression of the true branch e_2 or the expression of the false branch e_3 is evaluated; a call to a volatile function; a call to a built-in function; a reference to a cell area; or indexing a component of an array formula. An array formula expression ae is itself an expression which is expected to return an array.

We use the semantic sets and partial functions defined in figure 2.8 for describing entities of the spreadsheet and environment lookups. The *Number* set contains only proper numbers so not a number (NaN) values and infinities (∞, $-\infty$) are excluded from the set. The *Error* set contains error values such as `#DIV/0!` for division by zero. The set *ArrVal* contains the set of array values since they are first-class citizens in Funcalc. Conventional spreadsheet applications like Excel and LibreOffice Calc support arrays only using array formulas whose result

$$
\begin{array}{lcll}
n & \in & Number & = & \{ \text{ proper numbers } \} \\
 & & Error & = & \{ \texttt{\#DIV/0!}, \texttt{\#CYCLE!} \} \\
av & \in & ArrVal & = & \{ Av(w, h, [[v_{ij} \mid 1 \leq i \leq w, 1 \leq j \leq h]]) \} \\
ca & \in & Addr & = & \{ \text{ cell addresses, e.g. B2, G\$6 etc. } \} \\
v & \in & Value & = & Number + Error \\
e & \in & Expr & = & \{ \text{ formulas, e.g. } \texttt{=1+2} \} \\
\phi & & & \in & Addr \to Expr \\
\sigma & & & \in & Addr \to Value \\
\alpha & & & \in & Expr \to Value
\end{array}
$$

Figure 2.8 – Semantic sets and environment maps used in the spreadsheet semantics [6].

must be distributed over a suitably sized area. In contrast, we can construct and manipulate arrays inside single cells in Funcalc. Each array value Av has a width w, height h and elements v_{ij} within the bounds of the array. Note that array indices are one-based. The set of values $Value = Number + Error$ says that a value v is either a proper number in $Number$ or an error value in $Error$. The addition notation thus denotes the disjoint union of two semantic sets. The set $Addr$ contains valid cell addresses. The $Expr$ set contains the set of formula expressions. We have chosen to omit some additional error values (such as e.g. `#NAME!` for unknown function names) and some semantics value sets (such as strings) that are found in realistic spreadsheet programs. They are easily added to the semantics but afford no additional benefit for conveying the spreadsheet language and its extensions.

To describe and access the formulas of a spreadsheet, we use a function ϕ that maps cell addresses to formula expressions so that when $ca \in Addr$ is a valid address, $\phi(ca)$ is the formula in cell ca. The function is undefined for invalid cell addresses and blank cells, hence its partialness. The ϕ function is not affected by recalculation but by editing the spreadsheet. As a formula expression depends on the values of its cell references, ϕ models the dependencies of a spreadsheet.

We model the result of a recalculation using another partial function σ such that $\sigma(ca)$ is the value at cell address $ca \in Addr$. The σ function gets updated by each recalculation and is likewise undefined for invalid cell addresses but not for blank cells. We have a choice of how to

represent constants. They have no formula but do have a value so one could argue they should therefore be in σ. On the other hand, they are not actual formulas. Alternatively, we choose to view a constant such as `1` simply as a formula of a constant expression `=1` as in [6] at least for the purposes of the semantics because real spreadsheet programs would make the distinction.

The final partial function α maps expressions to values and models the array values from the evaluation of array formula expressions, i.e. $\alpha(ae)$, whose individual value elements are shared between the components of the array formula. For cases where an expression does not evaluate to an array, the result of α is undefined, hence why α is also partial.

The big-step or natural semantics for the spreadsheet language is given below and we elaborate each rule in turn. An evaluation judgement $\sigma, \alpha \vdash e \Downarrow v$ states that given σ and α, an expression e *may* evaluate to a value v. We later expand this judgement form to include costs in chapter 8. We stress that the judgement *may* evaluate to a value v because an expression may in general evaluate to many different values. The expression `=1/RAND()` for example may evaluate to both a number value or a division by zero error value `#DIV/0!` depending on the result of the call to `RAND`.

$$\frac{}{\sigma, \alpha \vdash \mathtt{n} \Downarrow n} \text{ (e1)} \qquad \frac{ca \notin dom(\sigma)}{\sigma, \alpha \vdash \mathtt{ca} \Downarrow 0.0} \text{ (e2b)}$$

$$\frac{ca \in dom(\sigma) \qquad \sigma(ca) = v}{\sigma, \alpha \vdash \mathtt{ca} \Downarrow v} \text{ (e2v)}$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1 \in Error}{\sigma, \alpha \vdash \mathtt{IF}(e_1, e_2, e_3) \Downarrow v_1} \text{ (e3e)}$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow 0.0 \qquad \sigma, \alpha \vdash e_3 \Downarrow v_3}{\sigma, \alpha \vdash \mathtt{IF}(e_1, e_2, e_3) \Downarrow v_3} \text{ (e3f)}$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1 \qquad v_1 \neq 0.0 \qquad \sigma, \alpha \vdash e_2 \Downarrow v_2}{\sigma, \alpha \vdash \mathtt{IF}(e_1, e_2, e_3) \Downarrow v_2} \text{ (e3t)}$$

$$\frac{0.0 \leq v < 1.0}{\sigma, \alpha \vdash \texttt{RAND}() \Downarrow v} \text{ (e4)} \qquad \frac{\sigma, \alpha \vdash e_i \Downarrow v_i \in \mathit{Error}}{\sigma, \alpha \vdash \texttt{F}(e_1, \ldots, e_n) \Downarrow v_i} \text{ (e5e)}$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1 \notin \mathit{Error} \quad \ldots \quad \sigma, \alpha \vdash e_n \Downarrow v_n \notin \mathit{Error}}{\sigma, \alpha \vdash \texttt{F}(e_1, \ldots, e_n) \Downarrow f(v_1, \ldots, v_n)} \text{ (e5v)}$$

$$\frac{\begin{array}{cc} (c_1, r_1) = ca_1 & (c_2, r_2) = ca_2 \\ (c_l, c_r) = sort(c_1, c_2) & (r_t, r_b) = sort(r_1, r_2) \\ w = c_r - c_l + 1 & h = r_b - r_t + 1 \end{array}}{\sigma, \alpha \vdash \texttt{ca}_1 : \texttt{ca}_2 \Downarrow Av(w, h, [[\sigma[c_l + i, r_t + j] \mid 1 \leq i \leq w, 1 \leq j \leq h]])} \text{ (e6)}$$

$$\frac{}{\sigma, \alpha \vdash ae[i, j] \Downarrow \alpha(ae)[i, j]} \text{ (e7)}$$

Figure 2.9 – Big-step semantics for the small spreadsheet language [6].

- Rule (e1) states that a constant expression evaluates to the value of that constant.
- Rule (e2b) states that a cell reference to a blank cell evaluates to zero since the address is not in the domain of σ. For simplicity, we disregard invalid cell addresses that usually evaluate to the `#REF!` error value.
- The corresponding rule (e2v) for non-blank cells states that if a cell address ca is in the domain of σ and σ maps ca to the value v, then the cell reference evaluates to v.
- Rule (e3e) handles the case where the conditional expression of an `IF` evaluates to an error. If the conditional expression e_1 evaluates to an error value v_1, then the error is propagated as the result and the argument expressions e_2 and e_3 are not evaluated. Propagation of error values is a common property of spreadsheets and the reason why error values are normally first-class values.
- Rule (e3f) states that if e_1 may evaluate to zero then the false branch is taken by evaluating e_3 to a value v_3 which is the result of the entire expression.

- Similarly, rule (e3t) states that if e_1 may evaluate to a non-zero value v_1 then the true branch is taken by evaluating e_2 to a value v_2 which is the result of the entire expression.

- Rule (e4) simply states that calling the intrinsic volatile `RAND` function produces a value v between 0.0 (inclusive) and 1.0 (exclusive).

- Rule (e5e) handles error propagation for built-in error-strict function calls. It states that if some argument expression e_i may evaluate to an error value v_i, then v_i is the result of calling `F`. If more than one argument evaluates to an error value, the rule does not specify which one is chosen as the result. Neither does the rule specify a specific evaluation order for the arguments. Some functions are not error-strict and operate on values that are expected to be errors. There is no separate rule for these types of functions here but we handle them in chapter 8.

- Rule (e5v) states that if the argument expressions $e_1, \ldots, e_n$ evaluate to non-error values then the expression results in calling the actual function f with the value arguments $v_1, \ldots, v_n$.

- Rule (e6) handles evaluation of cell area expressions. The premises unpack the column and row indices of the cell references ca_1 and ca_2 of the cell area expression. The *sort* function sorts the indices in ascending order which is necessary since any opposing set of corner cells suffices to describe a cell area, i.e. the expressions `A1:B2`, `A2:B1`, `B1:A2` and `B2:A1` all describe the same cell area. We use the sorted indexes to compute the width and height of the array value. The result may then evaluate to an array value of width w and height h whose $w \cdot h$ elements are looked up via σ by adding offsets to its top-left coordinates using the syntax $\sigma[c, r]$.

- The final rule (e7) has no premises and states that to access an element of an array formula expression ae at indices i and j, ranging over columns and rows respectively, one must first retrieve the resulting array value for ae via α and index that. The indices are one-based so the top-left cell of the array formula would contain expression $ae[1, 1]$. Indexing must produce an error value if the value $\alpha(ae)[i, j]$ is not an array value or the indices are out of bounds.

Note that we did not include the semantics of any dynamic indexing functions on purpose since Funcalc does not support such functions. A straightforward approach would be to pass the σ environment to the dynamic indexing function [23, sec. 1.8.6] so that it can look up values. With the defintions of the grammar, sets and semantics of the small spreadsheet language, we next formally define the consistency requirements for a recalculation.

2.9 Consistency Requirements

In section 2.7, we alluded to the fact that the purpose of recalculation is to bring the spreadsheet to a consistent state without providing any details. We now formalise these requirements, that must be met by any recalculation algorithm, sequential or parallel. The requirements rely on the ϕ, σ and α mappings of our small spreadsheet language defined in the previous section.

$$dom(\sigma) = dom(\phi) \tag{2.1}$$

$$\forall ca \in dom(\phi) \;.\; \sigma \vdash \phi(ca) \Downarrow \sigma(ca) \tag{2.2}$$

$$\forall ae \in dom(\alpha) \;.\; \sigma, \alpha \vdash ae \Downarrow \alpha(ae) \tag{2.3}$$

Requirement 2.1 states that the domains of ϕ and σ must be the same. We do not require that $dom(\alpha)$ is also equal to these domains as α is only valid for entire array formula expressions while both σ and ϕ are valid for ordinary formula expressions, although array formula element access such as $ae[1,3]$ is considered a formula expression. Requirement 2.2 states that for every cell address ca in the domain of ϕ, and thus also in the domain of σ by way of requirement 2.1, the evaluation of its formula $\phi(ca)$ must agree or be consistent with the value of $\sigma(ca)$. Requirement 2.3 is similar to requirement 2.2 but handles array formulas by requiring that all array formula expressions ae in the domain of α must agree with the corresponding value produced by the expression $\alpha(ae)$.

Note that the consistency requirements do not specify how recalculation must otherwise proceed, whether sequentially or in parallel, or in which order. From an end-user perspective, this means that he or she does not have to be concerned with how a recalculation algorithm proceeds as long as the spreadsheet is brought to a consistent state.

How do the consistency requirements account for cycles? We can assign a special error value `#CYCLE!` to the cell in which the cycle was discovered that is then propagated appropriately such that the spreadsheet eventually assumes a consistent state. This satisfies the first requirement trivially since it is unaffected and satisfies requirement 2.2 as it finds a value for each cell. However, consistency does not demand complete termination of recalculation. Requirement 2.3 is satisfied since any component of the array value produced by an array formula that results in a cycle will produce a `#CYCLE!` error.

In practice, we treat a cycle as an error and halt recalculation if one is discovered and do not use the `#CYCLE!` error value. This means that the spreadsheet is left in an inconsistent state. We accept this inconsistency as there is no other way for recalculation to proceed anyway and cycles are an exceptional case. Different approaches are employed by the sequential and parallel algorithms in the presence of cycles but they all leave the spreadsheet in an inconsistent state. Thus the consistency requirements should be viewed in the context of a normal, acyclic recalculation. Using a `#CYCLE!` error value is also problematic since it may litter the spreadsheet with these values if the cycle is part of a large number of cells in the spreadsheet.

Chapter 3

Background

In this chapter we define the concept of *dataflow*, its numerous different historical definitions and how it is related to both spreadsheets and functional languages. Dataflow was used to describe a new kind of hardware architecture with enticing promises of true instruction-level parallelism developed to rival the Von Neumann machine [18]. Work on dataflow systems and the SISAL programming language were the inspiration for the static partitioning algorithm of chapter 9. We then also briefly delve into the history of spreadsheets and conclude the chapter by surveying relevant spreadsheet research.

Additional historical material on dataflow is provided by Johnston et al. [30] and Whiting et al. [32], while Abraham et al. [14] and our literature review [4] contain broader overviews of spreadsheet research.

3.1 Dataflow

Dataflow systems typically consist of nodes containing operations. Nodes are connected by a set of edges along which information or tokens flow. A small example is shown in figure 3.1 for a specific calculation with four inputs x, y, w, z and one output. Providing inputs $x = 2$, $y = 3$, $z = 4$, $w = 5$ would give us the result 26.

This demonstrates how the dataflow model of computation is driven forward by the availability of input. Computation is constrained by dependencies so a node only executes or fires when all of its inputs are available and the output edge is not occupied by an output token. This is sometimes called the firing rule [30] and is different from the control

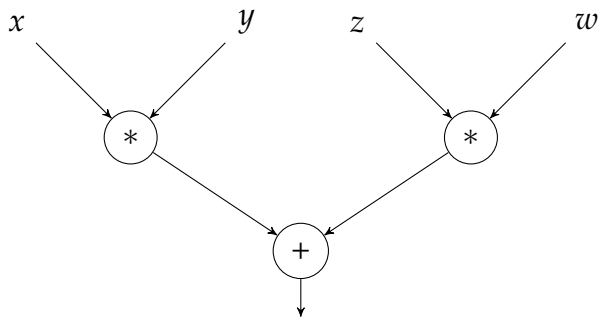


Figure 3.1 – A small example of a dataflow system with four inputs and one output. The two independent multiplications could be performed in parallel.

	A	B	C	D	E
1	2	3		4	5
2		=A1*B1		D1*E1	
3			B2+D2		

Figure 3.2 – Modelling the dataflow example of figure 3.1 in a spreadsheet. The inputs are cells A1, B1, D1 and E1.

flow associated with traditional imperative programming languages. It also demonstrates the natural capacity of such systems to express parallelism. For example, the two multiplications could be computed in parallel. Additionally, dataflow systems usually have no concept of shared state as information simply flows along edges and is transformed by the operations at nodes. Consequently, dataflow systems are typically side-effect free.

From the spreadsheet concepts of chapter 2, it is clear that the dataflow model is embedded in spreadsheets. More precisely, the support graph is essentially a dataflow graph and the firing rule is similar to how a cell can only be evaluated when all its dependencies have been evaluated. The spreadsheet in figure 3.2 models the small dataflow system of figure 3.1 with the inputs in cells A1, B1, D1 and E1.

Roughly speaking, dataflow can be divided into two categories based on whether they target low-level hardware or instructions (fine-grained) or software (coarse-grained) which led to their respective names.

3.1.1 A Brief History of Dataflow

The dataflow model can be traced back to 1966 [32]. *Fine-grained* dataflow dates back to the 1970's at MIT, with prominent researchers like K. Arvind and R. S. Nikhil. It was conceived in an attempt to design a new massively parallel hardware model that could surpass the Von Neumann machine model and avoid the so-called *Von Neumann bottleneck* [18, 30, 33]. The limiting factor was the bus shared between the central processing unit (CPU) and the memory system. Furthermore, it was argued that the global program counter and mutable memory made Von Neumann processors unsuitable for parallelism.

An example of fine-grained dataflow was the Id language that was compiled into a parallel machine language using dataflow graphs to model the flow of computations [33]. Hardware instructions were executed on a special Tagged Token Dataflow Architecture (TTDA) with data-driven instruction scheduling based on dependencies in the dataflow graph. Computations had to wait for their dependencies to complete, but otherwise independent computations could execute in parallel. The architecture was thus capable of remarkably fine-grained instruction-level parallelism. However, this level of granularity proved to be too fine and lead to the development of hybrid systems combining the Von Neumann and dataflow architectures [30].

Another example of fine-grained dataflow are so-called asynchronous or self-timed circuits [34]. While not as fine-grained as the work described in the previous paragraph, they exhibit similar properties since the circuits are not governed by a central clock, as in contemporary CPUs, but instead use signals to communicate akin to the flow of data that triggers the execution of instructions in the Id language.

John Hennessy, David Cann, Vivek Sarkar and others [19, 35, 36] worked on *coarse-grained* dataflow at the software level. Sarkar developed an optimising compiler for the first-order functional language SISAL which stands for Streams and Iteration in a Single Assignment Language. The compiler partitioned the program graph into groups that could be executed in parallel. A program graph is a type of dataflow graph where computation flows forward through the program as dictated by its control flow. SISAL was developed for large-scale scientific computing, an area dominated by Fortran at the time. Its development helped showcase that functional supercomputing was viable by obtaining comparable performance to Fortran programs on the Cray Y-

MP/864 supercomputer [19].

Yet another type of dataflow is called *synchronous* dataflow. Kahn [38] investigated this concept in 1974 with a formalisation of the semantics of a simplified language for communicating parallel processes modelled by a schema or network, closely resembling a dataflow graph. Halbwachs et al. [39] designed Lustre for reactive systems in 1991. They defined a dataflow model and a language which augments it with time-dependent flows, clocks and operators for constructing time-sensitive and event-driven programs. The synchronous aspect of Lustre was the formulation of conditions and relations using the semantics of the language that control the interplay of events.

In more recent times, Microsoft has released the Dataflow library [40] as part of the Task Parallel Library (TPL) where users build computations using various dataflow components called blocks. The entire computation is then initiated by supplying inputs to its input blocks and data is automatically passed through the system, in parallel if possible.

The Tensorflow project[1] is a framework by researchers at Google for accelerating machine learning algorithms on heterogeneous devices. The machine learning network or graph is constructed and started by providing input to the system. The dataflow model is then scheduled to run on a set of available devices such as CPUs, graphics processing units (GPUs) or tensor processing units (TPUs). TPUs are specifically designed hardware for the Tensorflow project.

3.2 Spreadsheet Background

3.2.1 A Brief History of Spreadsheets

One of the first visual and electronic spreadsheet application was VisiCalc, developed in the late 1970's by Dan Bricklin and Bob Frankston [8, 41]. It was written in assembler for the Motorola 6502 microprocessor used in the Apple II computer.

In the early 1980's, the Lotus 1-2-3 spreadsheet application was developed by Mitch Kapor [41] which introduced cell ranges, plots, database capabilities and other useful features. Later iterations of the Lotus spreadsheet software introduced further improvements and ideas [8]. The Forms/3 research spreadsheet application [42] strives to show how some

[1] `https://www.tensorflow.org/`

of the limitations of spreadsheets can be overcome. Examples are a limited number of types and a lack of abstraction. The authors show that concepts such as animated output, a dynamically sized grid of cells, and graphics as first-class types can be supported by the spreadsheet paradigm.

Today, the most well-known commercial spreadsheet application is perhaps Excel. It was originally developed for the 512K Apple Macintosh computer in 1984-1985 and set itself apart by using a graphical user interface and having mouse support [41]. Similar applications exist such as the web-based Google Sheets [43], the open-source LibreOffice Calc [44] and Gnumeric [45].

3.3 Spreadsheet Research

We mentioned a lack of academic involvement in the development of spreadsheets [15] in the introduction of this dissertation. Nevertheless, some research has attempted to transfer concepts to spreadsheets that have proved useful in programming languages. This section summarises some of the research related to the present work which was partly taken from the author's technical report "*A Literature Review of Spreadsheet Technology*" [4] and additional material not included in that report.

3.3.1 Functional Programming and User-Defined Functions

Most modern spreadsheet applications already allow users to define their own functions. For example, Excel permits user-defined functions in Visual Basic for Applications (VBA), and can interface with external languages. However, writing and debugging programs written in an external language requires end-users to learn a new programming language which is a large learning investment on their part. This section examines how research has tackled this issue.

There are different ways to combine functional programming and spreadsheets. One could embed an existing functional programming language in a spreadsheet application to enhance the formula language or carry over ideas and paradigms from functional languages. Conversely, other work has tried to use ideas from spreadsheets, like automatic recalculation and visual feedback, to improve the programming environment for functional languages.

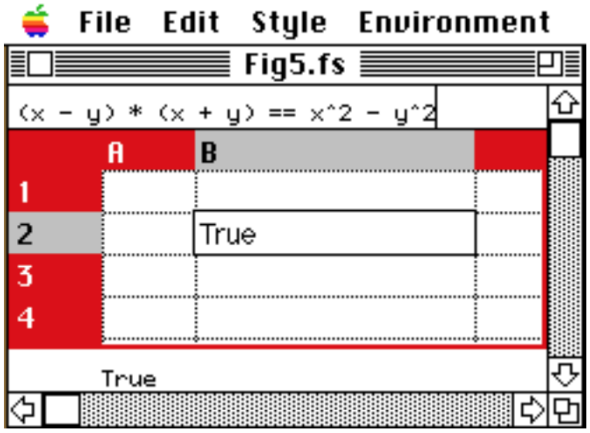
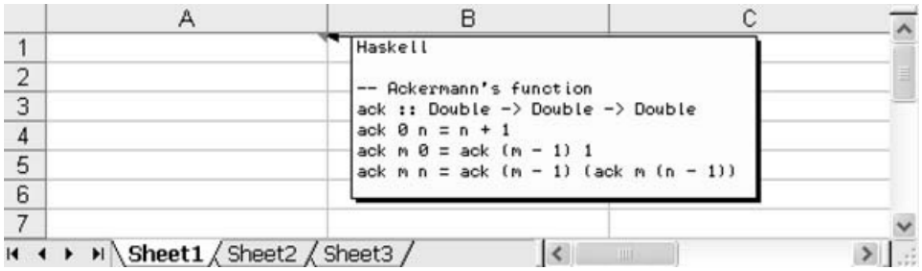


Figure 3.3 – Solving a symbolic algebraic equation using the symbolic evaluator in FunSheet [48].

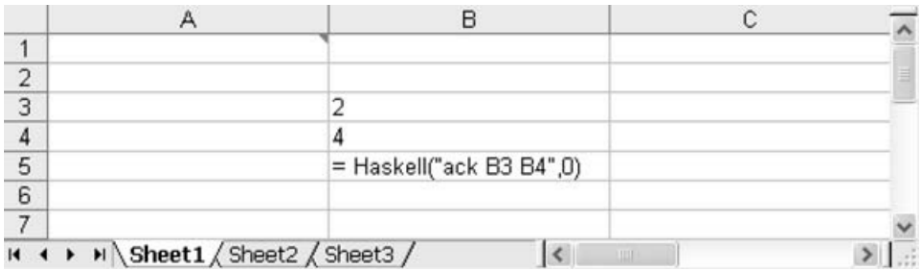
The concept of user-defined functions (UDFs) defined directly in spreadsheets were originally proposed by Peyton-Jones et al. [46] who postulated that functions defined in the spreadsheet paradigm would be a powerful feature for end-user software development. The proposal would let users define their own functions using a notation and an environment they are already familiar with, instead of resorting to unfamiliar languages such as VBA or C# which can be either slow or difficult to learn for end-users without a formal background in computer science [46]. Furthermore, they suggested that these functions could be automatically compiled behind the scenes at runtime to increase performance and support the same interactive edit-run cycle that users normally associate with spreadsheets when cells are modified. We will later see a concretisation of this idea in chapter 4 when we discuss the spreadsheet implementation we used in this thesis.

In a video presentation from 2009, Benfield [47] presents the Functional Model Deployment (FMD) framework for Excel which provides variable declarations, tuples and higher-order UDFs that can be referred to via handles. It also permits users to interface with external libraries, automatically generating the boilerplate VBA glue code needed for communication. Functions are evaluated using an evaluation function.

Hoon et al. [48] implement a spreadsheet application called FunSheet in the pure functional, lazy, higher-order language Clean [49] that features a symbolic evaluator for equations. For example, the symbolic evaluator will determine that the expression in figure 3.3 is true. Clean is



(a)



(b)

Figure 3.4 – Defining and using a Haskell function for Ackermann's function in a spreadsheet [13].

also used as the formula language of FunSheet. The application features over 60 predefined functions and users can define their own functions in a separate built-in function editor. Furthermore, each sheet has its own local function environment. The authors suggest using laziness to only update cells that are currently visible. As the user moves around, the cells that come into view can be updated on demand.

Wakeling [13] lets end-users call Haskell functions defined in Excel comments as shown in figure 3.4. Users call Haskell code using a appropriately named `Haskell` macro and the call is forwarded to an external Haskell interpreter. Users thus have access to a powerful functional language featuring higher-order functions, polymorphic types and lazy evaluation. These concepts may be hard for end-users to grasp.

Introducing powerful, expressive functional concepts to spreadsheets suffer from the fact that they demand a high learning cost from end-users. Such a concept is recursion. Casimir [10] claims that recursion is problematic because it may require many recalculations to iteratively compute a result but Yoder et al. [9] dispute this claim, arguing that natural-order recalculation, i.e. calculating a cell's dependencies before

itself akin to topological sorting, elegantly solves this problem. They do note though that the lack of formula-local variables can lead to considerable memory usage as intermediate values can only be stored in the "global" memory of the spreadsheet cells. We do not believe that this particular case of memory is a concern for modern spreadsheet applications and recursion does enable simple looping constructs. Peyton-Jones et al. [46] argue that the lack of inductive types hamper the expressiveness of recursion in spreadsheets compared to other functional languages.

Haxcel [50] instead enhances an external language with concepts from the spreadsheet paradigm. The interactive edit-run cycle and the immediate visual feedback is used in a programming environment for the pure functional language Haskell. Cells contain Haskell definitions that exist in different windows. Haskell is essentially the formula language for Haxcel's spreadsheet interface. An array library for high-level operations on arrays is also provided.

3.3.2 Object-Oriented Spreadsheets

Object-oriented programming has also been considered in the context of spreadsheets. Piersol [51] examines the Analytical Spreadsheet Package (ASP) which is entirely implemented in the dynamically typed Smalltalk-80 language. Cells in the ASP can hold any type that can be defined in Smalltalk-80 such as images, files or stacks. Automatic recalculation is achieved through notifications between objects in the cells of the spreadsheet, a feature already provided by Smalltalk-80.

McCutchen et al. [52] present a hierarchical spreadsheet model that supports flat tables, variable-sized lists and objects in effort to provide more structure to the otherwise unstructured grid of cells of most spreadsheets. Interestingly, their resulting Object Spreadsheets prototype tool is used for the development of interactive data-centric web applications. The spreadsheet serves as a data container for the application with object types organised in columns. Formulas can then manipulate this data e.g. via the dot notation found in traditional object-oriented languages. Regions of the spreadsheet are dedicated to displaying data and the tool supports procedures for updating data via the web application.

3.3.3 Array Programming

The two-dimensional grid layout of spreadsheets and copy-paste of relative cell references fit nicely with the array programming paradigm. Biermann et al. [53] developed a novel data structure called quad ropes to support fast concatenation and array parallelism in spreadsheets. Quad ropes are a combination of quad trees [54] used for efficient spatial queries and binary trees called ropes [55] with strings or arrays at its leaves which allow for constant-time concatenation. Quad ropes are thus parallelisable, capable of fast concatenation and can be rebalanced. They were implemented as the backing array representation in Funcalc [56] which we also use here to implement our tools. However, we do not use the version of Funcalc with quadropes in this work. Funcalc supports higher-order functions and first-class arrays both of which are useful for array programming. Biermann [57] provides more details on spreadsheet array programming.

3.3.4 Visualisation

Immediate visual feedback is already an invaluable feature of spreadsheets and work has been done to further extend its visual capabilities. Here, we do not refer to visualisation in terms of the graphical user interface (GUI) nor graphical elements such as plots or charts. We only discuss visualisation in terms of tools meant to provide users with an overview of the structure of a spreadsheet. This is especially crucial in organisations that possess large, complex spreadsheets that are both hard to manage and difficult to pass on to colleagues [58]. It is not uncommon that end-users inherit spreadsheets from co-workers within an organisation and must spend time deciphering the spreadsheet to understand its purpose.

Hermans et al. [58, 59] developed a tool to give a high-level dataflow diagram of the inter-worksheet relations of a spreadsheet and allow end-users to reason about its overall structure or help explain its purpose. The dataflow diagrams are annotated with arrows that depict inter-worksheet references whose thickness is proportional to the number of cell references between worksheets. The authors completed a series of empirical studies at the Dutch asset management company Robeco, and found that almost all of their interviewees grasped the dataflow diagrams and that there was an 80% consensus on the benefits of the tool

in their daily work. The authors also found that their tool helped give the recipient a better understanding of its layout and intended purpose when a spreadsheet was transferred between people. In later work, Hermans et al. enhanced the dataflow diagrams of the tool with data clone detection [60]. Here, clone clusters are identified as regions containing the same values as the result of copying formulas purely as data. Therefore, one region might contain values as a result of formula evaluation whereas its clones will contain just the pure values. Thus if the formulas are updated, the user must remember to manually update the copied values as well. Failing to do so may result in inconsistencies that in turn may lead to errors.

3.3.5 Parallelism and Performance

This section discusses both research that parallelises spreadsheet computation and increases performance through other means.

ActiveSheets [61] dispatches special plan files that describe inputs and work distribution in a spreadsheet to Nimrod [62], a framework for distributed computation. Custom VBA functions that represent user-defined simulations are executed in parallel by ActiveSheets. The custom functions send necessary data to the backend for evaluation. When retrieving results from Nimrod, ActiveSheets automates their aggregation and imports them into the spreadsheet which is normally required to be done by the user in Nimrod. ActiveSheets is capable of exploiting both intra- and inter-cell parallelism. Although two case studies are presented, no results are reported.

HPC Services for Excel [63] off-loads the evaluation of UDFs or entire workbooks to a Windows high performance computing (HPC) cluster using a service-oriented architecture (SOA). To run UDFs on the cluster, the user specifies an auxiliary file containing their definitions and dependencies. To run workbooks, a framework is available to let users define how independent calculations in a workbook can be partitioned and individual results merged.

In his 1996 dissertation, Wack [64] investigated parallelisation of spreadsheet programs using distributed systems and an associated machine model. The functional language Scheme is used as the spreadsheet language which introduces higher-order UDFs via lambdas. He partitioned and scheduled a set of predefined spreadsheet topologies and parallelised them via message-passing in a network of work stations.

Biermann et al. [22] rewrote cell arrays to calls to sheet-defined functions (SDFs), completely transparent to end-users. SDFs are a feature of Funcalc [56] and are higher-order UDFs defined directly in cells. We discuss them in more detail in chapter 4. More specifically, cell arrays were rewritten to array formulas of higher-order SDF calls on arrays such as map or prefix. Their approach parallelises the internal evaluation of each rewritten array but evaluates disjoint cell arrays sequentially.

In LibreOffice Calc [44] data-parallel expressions can be automatically compiled into OpenCL kernels that execute on AMD GPUs [65]. They reported a 500-fold speed-up for a single benchmark spreadsheet. Presentations from LibreOffice conferences also discuss multi-threaded execution of cell arrays [66] called formula groups in LibreOffice Calc.

Since Excel 2007, Excel features multi-threaded recalculation enabled via a checkbox in the settings. The exact number of processors desired can also be selected. Little information is available about how this process works internally[2].

Swidan et al. [21] manually refactor a spreadsheet performing a Monte Carlo simulation that would ordinarily take around 10 hours to recompute. Besides refactoring formula expressions etc., the authors also restructure the spreadsheet to off-load evaluation to a remote cluster.

Bøgholm et al. [67] generate timed automata (TAs) from spreadsheets which are then used in the UPPAAL-STRATEGO model checker[3] to schedule dataflow computations for parallel execution. Each cell is regarded as a separate task with different states signifying its progress e.g. *Waiting*, *Executing* or *Finished*, while the transitions of the TAs are annotated with preconditions that must be met in order to transition from one state to another. A model for the system's CPUs is also given. The model checker then finds the fastest possible trace for the model resulting in the shortest time taken to run all tasks according to a global clock. The paper also details a master thesis[4] [68] on a dependency scheduler for Funcalc. It takes a set of cells packaged as tasks and a description of their dependencies and dynamically tries to schedule them as efficiently as possible. The dependency scheduler achieved good speed-ups [67]

[2]This link provides a few details: `https://docs.microsoft.com/en-us/office/client-developer/excel/multithreaded-recalculation-in-excel`

[3]`http://people.cs.aau.dk/~marius/stratego/`

[4]Several master theses related to spreadsheets are also available at `https://www.itu.dk/people/sestoft/itu/specialer.html`

compared to sequential execution even beating the GPU-accelerated LibreOffice Calc in one case.

A patent [69] describes, in unusual detail, how formula expressions can be compiled to x86 machine code. The patent uses IEEE NaN values to encode spreadsheet error values since adding a non-NaN value and a NaN value results in a NaN value.

SpreadsheetGear [70] is a collection of commercial plugins for Excel, one of which is a recalculation engine that boasts multi-threaded recalculation. They parallelise spreadsheet recalculation via the TPL [71] but do not disclose how this is done or exactly how and if they replace Excel's existing recalculation engine.

The mechanics of spreadsheet recalculation are related to the idea of *self-adjusting computations* [72]. In his thesis, Acar describes a system that only needs to be partially updated if a modification only depends on a subset of the system, akin to minimal recalculation in spreadsheets. He also discusses how such computations can be done in parallel. Interestingly, this work inspired the OCaml `incremental` library [73] used by the Jane Street bank for constructing such self-adjusting computations.

In [31], Mokhov et al. study build systems such as the well-known `make` command line tool where the dependencies of a build system are compared to spreadsheets. For example, a modification of a file may invoke a partial rebuild of the system much like a cell update would. They also describe some of the inner workings of Excel's recalculation algorithm.

3.3.6 Spreadsheet Patents

Several spreadsheet patents have been filed although many of them tend to be purposefully quite vague. Their scope is extremely large so instead of summarising every single patent here, we refer interested readers to the comprehensive lists provided by [23] and [74]. The latter and most recent of those reports lists 598 spreadsheet patents in total between 2006 and 2018.

Chapter 4

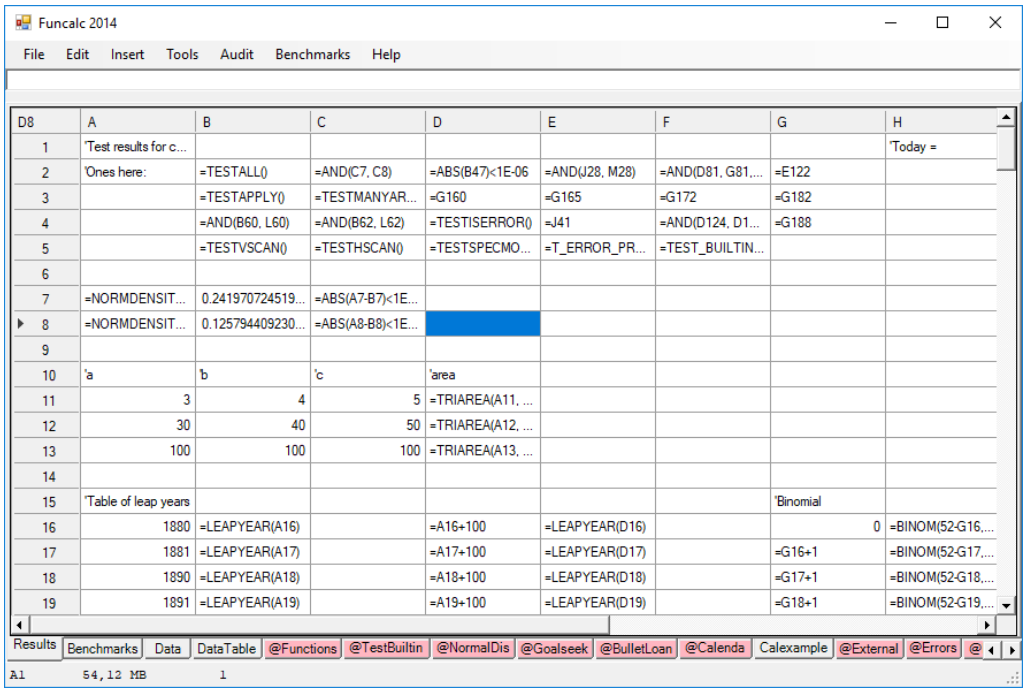
The Funcalc Spreadsheet Application

The present work was implemented in the Funcalc [56] spreadsheet application. It is primarily intended for research prototyping and written in C#. A screenshot of Funcalc is shown in figure 4.1. It was originally developed as Corecalc in 2006 and later extended in a master thesis under the name SupportCalc [75]. The extension tested ideas presented in [76] for improving the representation of the support graph. In 2009, Sestoft began developing SupportCalc into what is now Funcalc [77, 78] to showcase the implementation of compiled higher-order, user-defined functions defined directly in cells.

In this chapter we describe some specific implementation details of Funcalc useful in later discussions, some of which set it apart from other spreadsheet applications. The contents of this chapter is based on the book "*Spreadsheet Implementation Technology*" [23]. To ensure clarity, all code in this thesis is given as approximations of the real C# code with distracting details omitted.

4.1 Recalculation In Funcalc

As discussed in section 2.7, there are two major types of recalculation: *minimal* and *full* which we now describe in the context of Funcalc. First we show how cells and their state relate to recalculation.



D8	A	B	C	D	E	F	G	H
1	'Test results for c...							'Today =
2	'Ones here:	=TESTALL()	=AND(C7, C8)	=ABS(B47)<1E-06	=AND(J28, M28)	=AND(D81, G81,...	=E122	
3		=TESTAPPLY()	=TESTMANYAR...	=G160	=G165	=G172	=G182	
4		=AND(B60, L60)	=AND(B62, L62)	=TESTISERROR()	=J41	=AND(D124, D1...	=G188	
5		=TESTVSCAN()	=TESTHSCAN()	=TESTSPECMO...	=T_ERROR_PR...	=TEST_BUILTIN...		
6								
7	=NORMDENSIT...	0.241970724519...	=ABS(A7-B7)<1E...					
8	=NORMDENSIT...	0.125794409230...	=ABS(A8-B8)<1E...					
9								
10	'a	'b	'c	'area				
11	3	4	5	=TRIAREA(A11, ...				
12	30	40	50	=TRIAREA(A12, ...				
13	100	100	100	=TRIAREA(A13, ...				
14								
15	'Table of leap years						'Binomial	
16	1880	=LEAPYEAR(A16)		=A16+100	=LEAPYEAR(D16)		0	=BINOM(52-G16,...
17	1881	=LEAPYEAR(A17)		=A17+100	=LEAPYEAR(D17)		=G16+1	=BINOM(52-G17,...
18	1890	=LEAPYEAR(A18)		=A18+100	=LEAPYEAR(D18)		=G17+1	=BINOM(52-G18,...
19	1891	=LEAPYEAR(A19)		=A19+100	=LEAPYEAR(D19)		=G18+1	=BINOM(52-G19,...

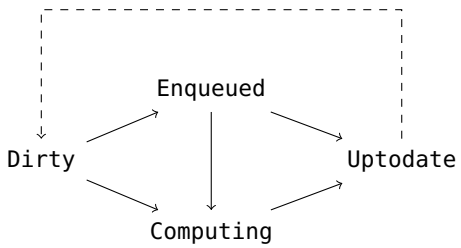
Figure 4.1 – A screenshot of Funcalc with an ordinary data sheet in focus. Formulas are being explicitly displayed instead of values. The pink tabs at the bottom are function sheets where SDFs can be defined.

4.1.1 Cell States

Cells are interpreted in sequential Funcalc and each formula cell has an internal state that is used by the interpreter during recalculation to keep track of their progress. We focus solely on formula cells here as constants involve no work and can be considered to always be computed. There are four states in total as shown in figure 4.2a that encode cell state. A cell is `Dirty` if it has not yet been computed; `Enqueued` if the cell is on the evaluation queue which is used in minimal recalculation; `Computing` if the cell is currently being computed; or `Uptodate` if the cell has been evaluated and its value is available. A state transition diagram is shown in figure 4.2b for reference in the following sections that explain minimal and full recalculation. Listing 4.1 shows the code for the `Formula` class. It contains an expression such as `=1+2`, a cell state and a value representing the result of evaluating the expression. Two properties are also provided for accessing the state and value.

```
1 public static class CellState
2 {
3     public const int
4         Dirty     = 0,
5         Enqueued  = 1,
6         Computing = 2,
7         Uptodate  = 3;
8 }
```

(a) The `CellState` class for representing different cell states.



(b) The possible state transitions of a cell.

Figure 4.2

```
1  public class Formula : Cell
2  {
3      private Expr expr;   // The formula's expression tree
4      private int state;   // Initially Dirty
5      private Value value; // Result from evaluating expr
6
7      // ...
8
9      public int State
10     {
11         get => state;
12         set => state = value;
13     }
14
15     public Value Cached
16     {
17         get => value;
18     }
19 }
```

Listing 4.1 – The class for representing formula cells.

4.1.2 Minimal Recalculation

Minimal recalculation evaluates cells in a breadth-first manner using an evaluation queue Q of pending work as shown in the overview in figure 4.3. It does so by starting at the recalculation roots and following the support graph to evaluate cells that need to be updated. The recalculation roots are first marked `Dirty` along with all reachable cells from the roots. Recall that this is the set of modified and volatile cells that must be updated to complete minimal recalculation. The recalculation roots are then enqueued in Q. A cell is popped from Q, evaluated, and its supported cells enqueued. Recalculation continues until Q is empty

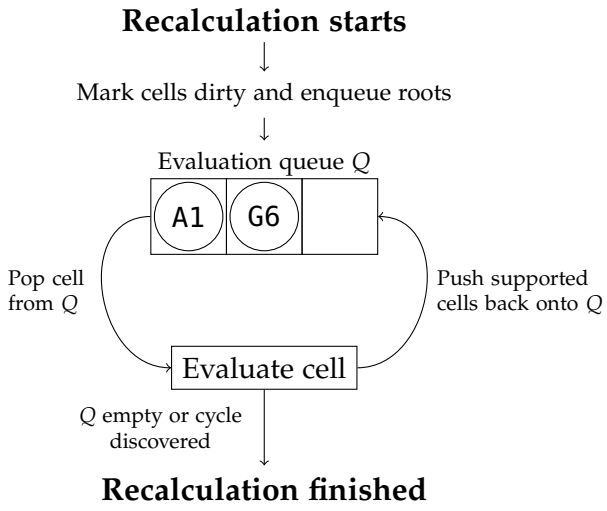


Figure 4.3 – Using a queue to compute cells in a breadth-first manner in sequential minimal recalculation. A cell is popped from the queue, evaluated, and then its support cells are enqueued. This continues until the queue is empty.

or a cycle is discovered. The code for minimal recalculation is shown in listing 4.2 with auxiliary functions in listings 4.3 and 4.4.

```
1  public class Workbook
2  {
3      public void RecalculateMinimal(List<Cell> roots)
4      {
5          foreach (Cell root in roots) {
6              MarkDirty(root);
7              Enqueue(root);
8          }
9
10         while (Q.Count > 0 && !CycleFound()) {
11             Cell cell = Q.Dequeue();
12             Eval(cell);
13         }
14     }
15 }
```

Listing 4.2 – Code for a minimal recalculation.

Function `RecalculateMinimal` accepts a list of recalculation roots. First, each root and all cells reachable from the roots are marked `Dirty`. The `MarkDirty` method (lines 1-10 in listing 4.3) marks an `Uptodate` cell `Dirty`, and its supported cells are recursively marked `Dirty` as

well. In other words, all cells that are transitively reachable from that cell in the support graph. This state change is denoted by a dashed line in the state transition diagram of figure 4.2b. The roots are then enqueued in the evaluation queue `Q`. The `Enqueue` method (lines 12-18 in listing 4.3) enqueues only `Dirty` cells. It changes a cell's state from `Dirty` to `Enqueued`, so that each cell is only enqueued once.

```
1  public void MarkDirty(Cell cell)
2  {
3      if (cell.State != CellState.Dirty) {
4          cell.State = CellState.Dirty;
5
6          foreach (Cell supp in cell.SupportedCells()) {
7              MarkDirty(supp);
8          }
9      }
10 }
11
12 public void Enqueue(Cell cell)
13 {
14     if (cell.State == CellState.Dirty) {
15         cell.State = CellState.Enqueued;
16         Q.Enqueue(cell);
17     }
18 }
```

Listing 4.3 – Code for marking cells dirty and enqueueing them used in minimal recalculation.

Afterwards, the `RecalculateMinimal` method enters a loop that continues as long as `Q` is not empty, and a cycle has not been found via the `CycleFound` method. If a cycle is found so `CycleFound` returns true but the cycle is subsequently removed by the user, `CycleFound` is reset to return false. The body of the loop pops a cell from the queue and evaluates it by calling `Eval` which is defined in listing 4.4. The `Eval` method first checks if the cell's state is `Computing`. If it is, computing this cell's value depends on itself creating a cycle. An exception is thrown to halt recalculation.

We evaluate a cell if its state is either `Dirty` or `Enqueued`. In the case of `Dirty`, we reached the cell via a dependency as it would otherwise have been evaluated when traversing the support graph where it would have been marked as `Enqueued`, covering the second case. To evaluate the cell, we set its state to `Computing`, evaluate its expression using `EvalExpr` and set its state to `Uptodate`. The `EvalExpr` method recursively evaluates a cell's dependencies via any cell references in its

```
1  public class Interpreter
2  {
3      public Value Eval(Formula cell)
4      {
5          switch (cell.State) {
6              case CellState.Computing:
7                  throw new CyclicException(...);
8
9              case CellState.Dirty:
10             case CellState.Enqueued:
11                 cell.State = CellState.Computing;
12                 cell.EvalExpr();
13                 cell.State = CellState.Uptodate;
14
15                 if (UseSupportSets) {
16                     foreach (Cell supp in cell.SupportedCells()) {
17                         Enqueue(supp);
18                     }
19                 }
20                 break;
21
22             case CellState.Uptodate:
23                 break;
24         }
25
26         return cell.Cached;
27     }
28 }
```

Listing 4.4 – Code for evaluating a cell.

expression. This is how cycles are detected as we can follow dependencies back to the same cell again in the presence of a cycle. The cell's support set is then enqueued in `Q` if the global flag `UseSupportSets` is true which is always the case for a minimal recalculation. The flag is useful for full recalculation as we will see shortly, and for one of the extensions to the static partitioning algorithm of chapter 9.

In the final case, the cell is already `Uptodate` and we can return its value via the `Cached` property to get the cell's most recent value. A cell is already `Uptodate` if it is a dependency of some cell that is in the transitive closure of a recalculation root.

4.1.3 Full Recalculation

In a full recalculation, all cells in the workbook are marked `Dirty` and `Eval` is iteratively called on all cells in an arbitrary order without using a queue or the support graph. The code for a full recalculation is shown

in listing 4.5. Here, we set `UseSupportSets` to false so supported cells are not enqueued.

```
1  public class Workbook
2  {
3      public void RecalculateFull()
4      {
5          // Mark all cells dirty
6          foreach (Sheet sheet in this) {
7              foreach (Cell cell in sheet) {
8                  cell.State = CellState.Dirty;
9              }
10         }
11
12         UseSupportSets = false; // Do not use support sets
13
14         // Evaluate all cells
15         foreach (Sheet sheet in this) {
16             foreach (Cell cell in sheet) {
17                 Eval(cell);
18             }
19         }
20     }
21 }
```

Listing 4.5 – Code for performing a full recalculation.

4.1.4 Full Recalculation and Support Graph Rebuild

There is a third type of recalculation which discards the old support graph and rebuilds it, then performs a full recalculation to ensure that all cells are `Uptodate`. This type of recalculation is used when a spreadsheet is initially loaded and its support graph is not yet built.

4.1.5 Dynamic Indexing Functions

Although Funcalc does not support any dynamic indexing functions, we briefly discuss how they could be implemented here. When evaluating a cell containing a call to `INDIRECT`, its expression and dependencies are evaluated as usual. Since the cells referred to by `INDIRECT` are only known at evaluation time, we cannot create support edges to a cell containing a formula that contains a call to `INDIRECT`. Consequently, they may never get enqueued in the evaluation queue which could lead to inconsistencies. Alternatively, we could modify dependencies during evaluation but this would further complicate the algorithm and lead

	A	B
1	=DEFINE("triarea", B6, B2, B3, B4)	
2	"a="	2
3	"b="	3
4	"c="	4
5	"s="	=(B2+B3+B4)/2
6	"result="	=SQRT(B5*(B5-B2)*(B5-B3)*(B5-B4))

Figure 4.4 – An SDF for computing the area of a triangle. The function takes three parameters (cells B2, B3 and B4 in green), has one intermediate cell (B5 in light blue) and a single output cell (cell B6 in blue).

to concurrent modification in a parallel implementation. To avoid inconsistencies and static dependency analyses, we could use the same strategy as Excel by marking them as volatile which is also suggested by Sestoft [23, sec. 5.5]. As mentioned, this sacrifices minimality but is a simple and elegant solution for handling dynamic dependencies.

4.2 Sheet-Defined Functions

We have already touched upon UDFs in our survey of spreadsheet research in section 3.3 but SDFs are a slightly different concept. Funcalc supports SDF definitions in special function sheets. Figure 4.1 shows some function sheets in the tabs at the bottom of the screenshot denoted by a leading "@" symbol and a pink background. Function sheets also have a pink border to distinguish them from ordinary data sheets with a grey border. An SDF for computing the area of a triangle given its side lengths is shown in figure 4.4. The call to the intrinsic function `DEFINE` specifies the name of the function, followed by a single output cell and zero or more input cells. Input cells are highlighted with a green background, cells containing intermediate computations with a light blue background, and the output cell with a blue background. SDFs are automatically compiled to Common Intermediate Language (CIL) bytecode [79] when it is initially defined or the cells in its definition are changed. This supports the same edit-run cycle as users know from ordinary data sheets. The compilation process is described in more detail in [23].

SDFs greatly increase expressiveness since they support higher-order functions and tail-recursion. Funcalc also supports first-class arrays that make it more natural to work with such functions. For example, end-

	A	B
1	1	2
2	3	4
3	=MAP(CLOSURE("SIN"), A1:B2)	

(a)

C12	A	B	C	D
1	1	2		
2	3	4		
3	[[0.84147098480...			
4		[[0.841470984807897; 0.909297426825682]; [0.141120008059867; -0.756802495307928]]		
5				

(b)

Figure 4.5 – Using `MAP` on the cell area in A1:B2. The resulting array value is shown on the right.

users can use the `MAP` or `REDUCE` staples of functional programming languages. Other spreadsheet applications usually do not support first-class arrays. However, array formulas can be used to distribute the array into a suitably sized cell area. Figure 4.5 shows an example of using `MAP` to apply the sine function to each cell in a cell area and store the result in a cell. The `CLOSURE` function creates a function value from a function name.

SDFs have the same benefits as functions in conventional programming languages, providing modularity and *re-usable abstractions* instead of relying on error-prone copy-paste operations of formulas [46]. Consider the difference between using `TRIAREA` for computing the area of a triangle versus using and copy-pasting a formula in figure 4.6. Triangle side lengths are defined in cells A2, B2 and C2. It is harder to tell what is being computed with the formulas even with headers unless we are intricately familiar with the mathematical formula for computing the area of a triangle. Moreover, copy-paste operations are the source of several errors in spreadsheets [24, 60, 80, 81]. By assigning a name to a function it also becomes more self-documenting. In this case, we also eliminate an intermediate computation which can be moved inside the `TRIAREA` function. Besides support for SDFs, Funcalc also provides a number of predefined built-in functions defined in C#.

4.2.1 External Libraries

Funcalc provides the `EXTERN` function for calling C# methods defined in a dynamic link library (DLL). We used `EXTERN` to call some of C#'s string manipulation methods when translating Streams and Iteration in a Single Assignment Language (SISAL) functions to Funcalc [7]. For example, the expression

```
=EXTERN("System.String.Substring(II)T", "funcalc", 3, 4)
```

	A	B	C	D	E
1	"a"	"b"	"c"	"s"	"area"
2	3	4	5	=(A2+B2+C2)	=SQRT(D2*(D2-A2)*(D2-B2)*(D2-C2))

(a)

	A	B	C	D
1	"a"	"b"	"c"	"area"
2	3	4	5	=TRIAREA(A2, B2, C2)

(b)

Figure 4.6 – Computing the area of a triangle with side lengths 3, 4 and 5 using formulas versus using an SDF. The SDF eliminates an intermediate computation and can be readily reused.

would call the `Substring` method of C#'s string class on the string "funcalc" with index 3 and length 4, returning the string "calc". Single characters are used to denote argument and return types. The `I` characters in parentheses denote the two integer argument types and the `T` character after the parentheses is the string output type.

4.3 Comparison With Other Spreadsheet Applications

Funcalc is intended for research prototyping and is not a commercial product. Therefore, it has some limitations when compared to commercial or open-source spreadsheet applications such as Excel or LibreOffice Calc that are intended for real-world use. For example, Funcalc does not support structured entities such as pivot tables or embedded media like plots and images, although plots could be elegantly combined with SDFs which could act as the input function for plotting.

For these reasons, we do not believe it is particularly meaningful to compare the evaluation times of Funcalc with its real-world counterparts. Regardless, there have been several occasions where such a comparison has been requested by reviewers or peers so we have provided comparisons in appendix B between Funcalc and Excel for readers who may be interested.

Now that readers are familiar with core spreadsheet concepts and Funcalc, we move on to the first part of the thesis concerning parallelisation of Funcalc's cell interpreter.

Part I

Dynamic, Parallel Spreadsheet Interpreters

Chapter 5

A Task-Based Parallel Cell Interpreter

The contents of this chapter is taken from the paper "*Puncalc: task-based parallelism and speculative reevaluation in spreadsheets*" [1, 2] that is joint work with Florian Biermann.

The first part of the dissertation presents a parallel, task-based algorithm for *minimal* recalculation which *dynamically* exploits *local* parallelism during evaluation. Whenever a user modifies a cell, one can picture an acyclic graph of dependencies to be recalculated in response to the update, with the modified cell as its root. Such a graph is amenable to parallel evaluation, visualised in figure 5.1. The forwards flow of data along the support edges is constrained by dependencies but otherwise independent computations can proceed in parallel. Colours denote different threads.

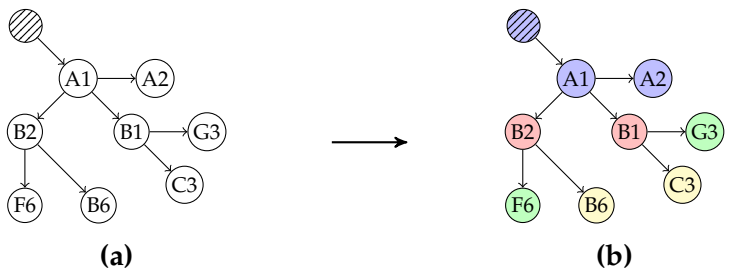


Figure 5.1 – An example of a computation graph of cells that can be recalculated in parallel. The recalculation originates at the striped cell, e.g. it could have been modified by the user. Only support edges are shown for clarity.

We require three things of a parallel algorithm for minimal recalculation. First, it should automatically attempt to evaluate cells in parallel once initiated. Second, it should be largely agnostic to the topology of a spreadsheet. Third, we must be able to detect cycles in parallel.

The rest of the chapter is structured as follows. Section 5.1 discusses the changes necessary to make Funcalc thread-safe as its original implementation is entirely sequential. The alterations we present are used as a foundation for the other parallel algorithms presented later. Section 5.2 presents the core ideas behind parallel, minimal recalculation and its implementation. It builds on the sequential cell interpreter from chapter 4 also using a global queue to evaluate cells in a breadth-first manner. We discuss the choice of using tasks for parallel evaluation, and a new termination condition, since the size of the queue alone is no longer sufficient. Section 5.3 discusses our implementation of parallel cycle detection. An optimisation to the algorithm is explained and implemented in section 5.4 that exploits sequential dependencies of cells. Section 5.5 examines the cycle detection algorithm's correctness and examines it in the light of the consistency requirements on recalculation. We present results on several benchmark spreadsheets in section 5.6 and make observations on them in section 5.7. The final section covers a race condition we found after the development and benchmarking of the algorithm. We show how it can be triggered and how it violates the consistency requirements we set forth in section 2.9. While this is unfortunate, we present an improved parallel interpreter in chapter 7 that eliminates this race. The conditions under which it manifests are not all present in our benchmark spreadsheets.

5.1 Thread Safety In Funcalc

Although Funcalc is a higher-order functional language, its implementation uses mutable state to make it efficient. Therefore, several components of Funcalc must be made thread-safe: access to cell state; access to a cell's cache containing the most recent result of an evaluation; the internal SDF compiler; and the global queue which is also used in the parallel variant of minimal recalculation.

We first discuss cell state and the cell cache in section 5.1.1 followed by a section on the compiler in section 5.1.2. A simple optimisation that works well in a sequential setting is discussed in section 5.1.3 and we

argue why this optimisation is not suited for a parallel context. Thread-safety of the global queue is discussed when we present the parallel minimal recalculation algorithm in section 5.2.

5.1.1 Thread-Safe Cell State and Cell Cache

Threads will inevitably race to manipulate cells and their state. To establish some intuition for how our approach makes this thread-safe, consider the following. We let threads compete for setting a cell's state to `Computing`. Whichever thread successfully sets the state proceeds to evaluate the cell while other threads must wait for the cell to be evaluated and then read its value from the cache.

How do we ensure that threads can set the state and access the cache in practice? We could use locks to implement the above scheme but as it turns out, locking comes at the cost of performance and correctness. Threads that wait for locks can be de-scheduled by the operating system incurring a performance cost if a cell's value becomes available sooner rather than later. The cells in the benchmark spreadsheets we use have a relatively low average computation time which makes de- and re-scheduling costly. In terms of correctness, cyclic references can lead to deadlocks if we are not careful.

Instead, we implement concurrent access to cell state using compare-and-swap (CAS) [82, sec. 5.8]. A call `Cas(ref location, a, b)` compares `b` with the value stored at `location`. If they are identical, `a` is stored at `location` and the operation returns the previous value stored at `location`. Otherwise, another thread has updated the value at `location` and the operation fails. This leaves `location` unmodified and returns the value stored there. Regardless of the outcome, the entire operation is atomic.

```
1 do {
2     int old = ReadLocation();
3 } while (Cas(ref location, 100, old) != old);
```

Listing 5.1 – An example of using CAS to concurrently update a shared variable `location`.

Listing 5.1 shows typical use of CAS. One reads some value at `location` and updates `location` with a new value (100 in this case) via CAS. By

checking the return value of the CAS, we are able to tell if the update was successful or not. Therefore, this is typically done in a loop in lock-free code to retry the update until we succeed, i.e. if another thread concurrently modified the location between reading its value and calling CAS, a different value is returned than the one we read and the loop is retried.

To ensure visibility of updates to the state and cache of cells, they are now read as `volatile`[1] variables as shown in listing 5.2. Volatile reads and writes have acquire-release semantics that restrict the compiler's freedom in reordering reads and writes. Threads that fail to set a cell's state call the `Cached` property and spin until it is `Uptodate` in line 14 of listing 5.2.

```
1  public class Formula : Cell
2  {
3      private int state;
4      private Value value;
5
6      // ...
7
8      public int State
9      {
10         get => Thread.VolatileRead(ref state);
11         set => Thread.VolatileWrite(ref state, value);
12     }
13
14     public Value Cached
15     {
16         get => {
17             while (State < CellState.Uptodate) {
18                 // Spin until cell has been evaluated
19             }
20
21             return value;
22         }
23     }
24 }
```

Listing 5.2 – Ensuring thread-safe access a cell's state and cache.

The overall idea is thus to let threads compete for setting a cell's state using CAS. Whichever thread wins proceeds as described above while the other threads wait for the cell to be evaluated. We expand on this idea in section 5.3 in order to detect cycles in parallel.

[1]The C# keyword, not to be confused with cell volatility.

	A	...	K
1	100		
2	=A1	···	=A1
⋮	⋮	⋱	⋮
1000	=A1	···	=A1

Figure 5.2 – Cell A1 supports the large cell area A2:K1000. Using a `SupportArea` instead of representing all cells in the supported area individually saves memory.

5.1.2 Thread-Safe Sheet-Defined Function Compiler

Funcalc's SDF compiler framework is quite complex. As the compiler is not the main focus of this work, we settled for a simple solution by using a single global lock to control access to the compiler. This may seem like a potential performance bottleneck but SDFs are only compiled when the user modifies a cell that is part of its definition or when the spreadsheet is initially loaded. Even when many cells depend on an SDF, Funcalc performs a full recalculation in response to the update. A full recalculation is performed since a cell's dependence on specific SDFs is not actively tracked. We can also parallelise full recalculation as discussed in section 5.7. Therefore, using a single global lock is not problematic.

5.1.3 Thread-Safe Support Area

Funcalc's support graph is not represented by an actual graph but implicitly as the support set of each cell. In case a cell supports a large area of cells, it instead maintains a *support area* to avoid representing each cell in the supported area explicitly. This is depicted in figure 5.2 for cell A1 that supports the contiguous cell area A2:K1000.

Funcalc maintains a list of already visited support areas in descending order of size. This is used when marking cells `Dirty` to avoid redundant work that could lead to quadratic work in the worst case [23, sec. 4.3.4]. It is also used during sequential recalculation when a support area is to be enqueued in the global queue. Here, the list is first consulted to see if part of or the entire area has already been visited. The list is maintained in descending order of size to list larger areas first that may encompass more support areas. New support areas are inserted into the list according to their size. For example, if cell B1 in

figure 5.2 contained a value and the expressions of the cells in A2:K1000 where changed to `=A1+B1` we could avoid iterating over the large cell area a second time when all cells are `Uptodate`.

This optimisation works well for sequential recalculation but is more problematic for parallel recalculation since the list must be thread-safe. We experimented with using locks, CAS and other C# concurrent data structures but they all incurred a performance hit. Although we did not implement it due to time constraints, a concurrent skip list [82, chap. 14] might be suitable because it is self-balancing and allows insertions at arbitrary points[2]. In the end, we decided to instead disable the optimisation for parallel implementations.

Unfortunately, we were not aware of this optimisation during the development of the parallel algorithms. Preliminary experiments showed that only two of our benchmark spreadsheets were affected by disabling the optimisation, effectively cutting their performance in half. This is regrettable but their performance on the highest number of logical cores was still in the range of a few seconds. A thread-safe solution is therefore important future work for a parallel implementation.

5.2 Task-Based Parallel Minimal Recalculation

We opted to use tasks for the parallel cell interpreter for two reasons. First, we can readily use the Task Parallel Library (TPL) [71, 83] in C# to easily spawn tasks. The TPL implements an efficient work-stealing threadpool for distributing work among threads. Spawned tasks are sent to the threadpool where a thread is assigned to evaluate it. Second, tasks are designed to be a light-weight unit of work that seem ideal for evaluating many cells in parallel, and frees us from manual thread management.

We must address two problems when implementing a parallel recalculation algorithm: we must choose adequate termination criteria as the size of the queue alone is no longer sufficient; and second, we must detect and handle cycles correctly. We address the first problem in the following paragraphs. The second problem is addressed in section 5.3.

An overview of the parallel, task-based interpreter is shown in figure 5.3 which is similar to the sequential implementation in chapter 4

[2]The .NET framework does not provide a concurrent skip list implementation as of this time of writing.

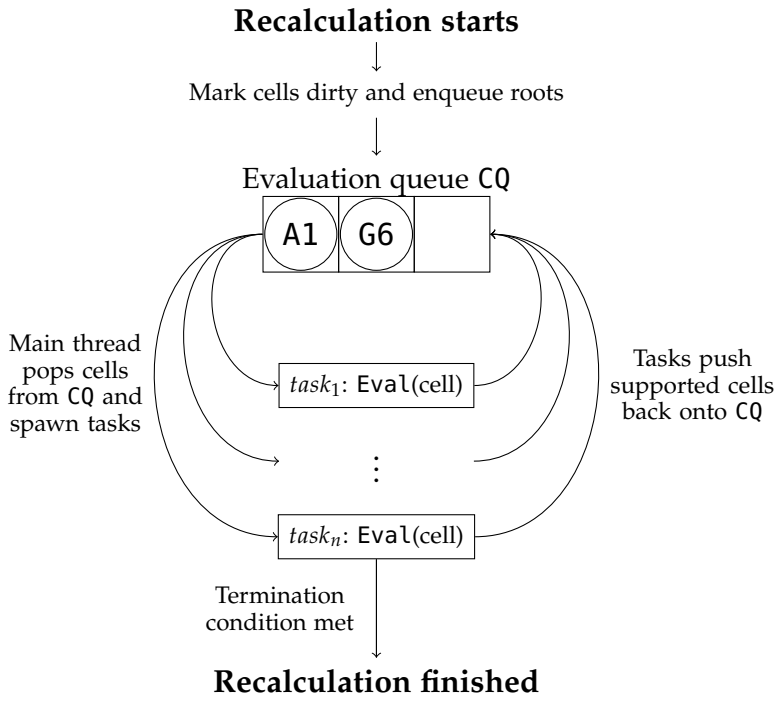


Figure 5.3 – Overview of task-based parallel, minimal recalculation. The main thread pulls work from the concurrent evaluation queue `CQ` and spawns tasks to evaluate each cell in parallel. The tasks push supported cells back onto the queue in parallel until a termination condition is met.

with some notable differences. To let threads concurrently push and pop cells from the queue we replace the sequential queue `Q` with a concurrent queue `CQ`, more specifically C#'s `ConcurrentQueue` class. The main thread pops cells from the concurrent queue and spawns tasks to compute each cell in parallel. Once finished, tasks push supported cells back onto the queue.

Listing 5.3 shows the main loop of parallel minimal recalculation corresponding to the overview in figure 5.3. Only the main thread executes this code, and so only it dequeues cells from `CQ` and spawns tasks. On the other hand, tasks concurrently push cells onto `CQ`. Parallel minimal recalculation begins like its sequential counterpart by marking all cells `Dirty` that are transitively reachable from the recalculation roots, and enqueuing the recalculation roots in `CQ`. The revised termination condition is necessary since the queue may be empty while there are still cells being evaluated in parallel so we cannot rely on the size of the queue

alone. To keep track of the number of cells being evaluated in parallel, we use an atomic counter inspired by Java 8's `LongAdder` implementation [84]. We allocate a new atomic counter and initialise it to zero in line 10.

```
1  public class Workbook
2  {
3      public void RecalculateMinimalPar(List<Cell> roots)
4      {
5          foreach (Cell root in roots) {
6              MarkDirty(root);
7              Enqueue(root);
8          }
9
10         LongAdder counter = new LongAdder(0);
11
12         while ((counter.Value > 0 || CQ.Count > 0) && !CycleFound()) {
13             Cell cell = CQ.TryDequeue();
14
15             if (cell != null) {
16                 counter.Increment();
17
18                 Task.Run(() => {
19                         Eval(cell);
20                         counter.Decrement();
21                         });
22             }
23         }
24
25         if (CycleFound()) {
26             // Report error to user...
27         }
28     }
29 }
```

Listing 5.3 – Code for parallel, task-based minimal recalculation.

The condition in the main loop in line 12 has changed to reflect our new termination condition. The loop runs as long as (1) there are cells being evaluated in parallel i.e. the value of the atomic counter is positive; or (2) there are still cells in `CQ`; and (3) a cycle has not been discovered. The ordering of the conditions is important since there are subtle timing issues that could otherwise make recalculation terminate prematurely. We discuss these subtleties after describing listing 5.3.

In the body of the loop in lines 12-23, the main thread tries to dequeue a cell from `CQ` which succeeds if the returned cell is non-null. We increment the counter to signal that a cell is being computed in parallel and spawn a new task to evaluate it. Cycle detection happens in

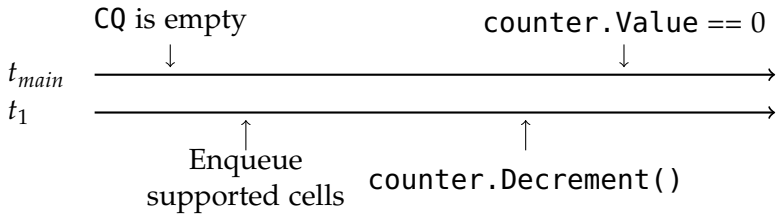


Figure 5.4 – A timeline, showing the interleaving of actions that could cause premature termination of the algorithm if the order of the first two termination conditions was reversed. Thread t_{main} is the main thread and thread t_1 is a TPL threadpool thread.

the `Eval` method implemented in section 5.3. Once the task has finished computing the cell, the counter is decremented since there is now one less cell being evaluated in parallel. In the absence of cycles, the queue will eventually become empty, any remaining tasks will finish evaluating their cells and the loop terminates, completing minimal recalculation. If a cycle is discovered, a global thread-safe flag is set which is returned by the `CycleFound` method.

Let us return to the subtleties of the termination condition we mentioned earlier. Imagine we swapped the first two termination conditions (1) and (2) that check the value of the counter and the size of the queue, respectively. Now consider the timeline in figure 5.4 where the queue is initially empty and the value of the counter is one. The main thread t_{main} would evaluate the loop condition and first see that `CQ` is empty. Before t_{main} continues, t_1 finishes evaluating a cell and enqueues a non-empty support set such that `CQ` is not empty anymore, but t_{main} does not see this. Thread t_1 then decrements the counter and t_{main} then observes that the value of the counter is zero. The result is that t_{main} now incorrectly believes that there are no cells currently being evaluated in parallel and that the queue is empty, exiting the loop prematurely. This subtle timing does not occur when we order the checks as in listing 5.3. Next, we can discuss how to detect cycles in parallel.

5.3 Parallel Cycle Detection

To detect cyclic dependencies during sequential recalculation, it was sufficient to inspect a cell's state and check whether it was `Computing`. We detect cycles through cell dependencies so this indicates we have en-

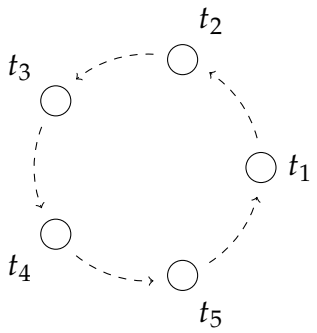


Figure 5.5 – A fundamental problem with detecting cycles in parallel. An arbitrary number of threads may become stuck in a cycle unless we employ some resolution strategy for detecting and handling it.

countered the cell a second time. Detecting cycles in parallel is less trivial due to two fundamental problems as depicted in figure 5.5 where five threads $t_1, \ldots, t_5$ are waiting for each other in a cycle.

First, even if a thread discovers a cell that is `Computing`, it is only a true cyclic dependency if the thread itself previously attempted to compute it, and subsequently rediscovered the cell through its dependencies. Flagging the discovery as a cycle would thus potentially report a false positive. Currently, a thread has no way of knowing who the owner of a `Computing` cell is so we need to establish some kind of cell *ownership*.

Second, even if we could check the owner of a cell, we would still need a resolution strategy to handle cases such as in figure 5.5 where the cycle consists of multiple threads. There must be a resolution strategy in place to discover the cycle in such cases and to avoid the threads becoming stuck indefinitely.

We could circumvent the problem entirely by statically checking for cycles before initiating a parallel recalculation, but this would defeat the purpose of recalculating in parallel in the first place. Such a check would also be too conservative e.g. in the case of `=IF(RAND()<0.5, A1, B1)` where one branch results in a cycle but the other one does not. In his dissertation on parallel message-passing in spreadsheets Wack [64, sec. 2.8.3] simply disallowed cycles altogether by not considering spreadsheet programs with cycles.

In conclusion, we need two things for detecting cycles in parallel. We need a concept of cell *ownership* so that a cyclic dependency is discovered

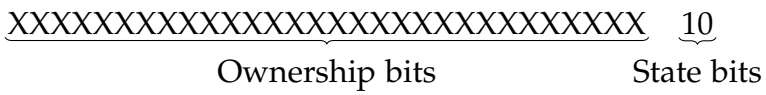


Figure 5.6 – Encoding ownership in a 32-bit integer. The 30 most significant bits encode the ownership bits and the 2 least significant bits encode the `Computing` state (2).

if a thread t_i encounters a cell that is `Computing` and whose *owner* is t_i. If a cell is `Computing` but owned by another thread, t_i waits until the cell becomes `Uptodate` before reading the cell's cached value.

We also need a tie-breaker, a resolution strategy that allows progress when threads are stuck in a cycle. Our strategy is called *speculative reevaluation* where threads are allowed to claim ownership of cells already owned by other threads under certain conditions. This will eventually allow some thread to claim ownership of enough cells to discover the cycle but we defer a detailed discussion until section 5.3.2. For example, t_3 in figure 5.5 could claim cells from the other threads and eventually discover itself. We tackle these two in the following two sections.

5.3.1 Encoding Ownership in Cell State

We encode ownership and cell state together so we can manipulate both properties using a single atomic CAS to avoid adding logic for handling partial states. The four cell states are represented by a 32-bit integer in the `CellState` class but their representation requires only 2 bits, leaving us with 30 bits to encode ownership as shown in figure 5.6. The ownership bits are only set and relevant when the cell is owned by a thread and its state is `Computing` and are zero for the remaining states. Three new utility methods for encoding and decoding ownership and state are implemented in the `CellState` in listing 5.4.

Method `DecodeState` accepts an encoded state and uses a binary `AND` operation with the combined bitmask of all non-zero states, to zero out everything but the two least-significant state bits. Method `DecodeOwner` also accepts an encoded state but returns the ownership bits by right-shifting away the two state bits. Finally, `EncodeOwner` retrieves the thread ID of the current thread, left-shifts it to make room for the `Computing` state and encodes them using a binary `OR` operation.

```
1  public static class CellState
2  {
3      public const int Dirty     = 0,
4                       Enqueued  = 1,
5                       Computing = 2,
6                       Uptodate  = 3,
7                       BitMask = Enqueued | Computing | Uptodate;
8
9      // How much to right-shift when removing state bits
10     private const int Shift = 2;
11
12     public static int DecodeState(int encodedState)
13     {
14         return encodedState & BitMask;
15     }
16
17     public static int DecodeOwner(int encodedState)
18     {
19         return encodedState >> Shift;
20     }
21
22     public static int EncodeOwner()
23     {
24         return (Thread.Id << Shift) | Computing;
25     }
26 }
```

Listing 5.4 – Updated code for the `CellState` class.

5.3.2 Speculative Reevaluation

With an understanding of the overall parallel recalculation algorithm and cell ownership, we can show the implementation of the `Eval` method called by each spawned task where cycles are detected using speculative reevaluation. The method is so named because threads can claim ownership of cells already owned, and reevaluate them speculatively based on the conjecture it has not yet been evaluated due to the presence of a cycle.

However, we have omitted one crucial detail until now. How do we decide which thread should be allowed to claim ownership over others? To decide, we impose a *precedence* order on threads. Thread IDs will serve our purpose nicely as they already impose a strict total order on threads. A strict total order is a binary relation $\prec$ on a set X where the following three properties hold for any $x_1, x_2, x_3 \in X$.

1. It is *irreflexive*: $x_1 \not\prec x_1$
2. It is *asymmetric*: $x_1 \prec x_2 \implies x_2 \not\prec x_1$

3. It is *transitive:* $x_1 \prec x_2 \wedge x_2 \prec x_3 \implies x_1 \prec x_3$

For our purposes, a thread t_i has precedence over t_j if $\texttt{Id}(t_i) < \texttt{Id}(t_j)$. If t_i and t_j form a cycle, then t_i is allowed to claim the cell from t_j in order to proceed and discover the cycle. Thread t_j cannot claim t_i's cell since it has a higher thread ID and thus lower precedence. The idea is that given any cycle of n threads, one thread will always have the highest precedence and eventually claim the other cells and discover the cycle.

Listing 5.5 shows the code for the `Eval` method. Like the sequential method, we read the cell's state and take actions accordingly. However here, we must decode the actual state from the encoded state.

If the cell is `Computing`, we decode the current owner of the cell by calling `DecodeOwner` and compare it to the current thread's ID. If they are identical, the current thread owns the cell and a cycle has been discovered. Notice that we set the cell's state to `Uptodate` in line 18 to allow any tasks waiting on the cell to complete using a stale value. We also set a global flag via the `SetCycleFound` method. This makes sure that `CycleFound`, called by the main thread, returns true so the main thread is notified. If the current thread's ID is less than that of the owner of the cell (line 20), we have precedence and so jump to the case for `Enqueued` to evaluate the cell. If we do not have precedence, we break out of the switch statement and call the `Cached` property to retrieve its value when the cell has been computed.

In case of `Dirty` or `Enqueued`, the cell is unclaimed and we can evaluate it by calling `EvalExpr` which now returns a boolean to signal whether or not we successfully claimed and evaluated the cell. If successful, the current thread enqueues the cell's support set, subject to the global `UseSupportSets` flag, and returns the cell's value via `Cached`. If the cell is already `Uptodate`, we can simply return its value.

The `Enqueue` method has been changed to enqueue cells in a thread-safe manner as multiple threads may attempt to enqueue a cell simultaneously if it is in the support set of more than one cell. We use a CAS to ensure that only one thread gets to enqueue the cell so that no cell is enqueued more than once.

Claiming Cell Ownership

Listing 5.6 shows the code for `EvalExpr` for claiming ownership of a cell and evaluating its expression. The current value of the cell is first

```
1  public void Enqueue(Cell cell)
2  {
3      if (Cas(ref cell.state, CellState.Enqueued, CellState.Dirty) ==
       ↪  CellState.Dirty) {
4          CQ.Enqueue(cell);
5      }
6  }
7
8  public class TaskBasedInterpreter
9  {
10     public Value Eval(Formula cell)
11     {
12         switch (CellState.DecodeState(cell.State)) {
13             case CellState.Computing:
14                 int tid = Thread.Id;
15                 int owner = CellState.DecodeOwner(state);
16
17                 if (tid == owner) {
18                     State = CellState.Uptodate;
19                     SetCycleFound(cell);
20                 } else if (tid < owner) {
21                     goto case CellState.Enqueued;
22                 }
23                 break;
24
25             case CellState.Dirty:
26             case CellState.Enqueued:
27                 if (cell.EvalExpr() && UseSupportSets) {
28                     foreach (Cell supp in cell.SupportedCells()) {
29                         Enqueue(supp);
30                     }
31                 }
32                 break;
33
34             case CellState.Uptodate:
35                 break;
36         }
37
38         return cell.Cached;
39     }
40 }
```

Listing 5.5 – Code for parallel evaluation of a cell.

read so we can later check whether another thread updated its value when we attempt to set it via CAS. There are three cases to consider: the cell is unclaimed, another thread has claimed the cell but we have precedence, or another thread has claimed the cell and we do not have precedence. Let us discuss each case in succession referring to listing 5.6 and the auxiliary methods in listing 5.7.

The first case is given by lines 9-10. We decode the state bits from

```
1  public class Formula : Cell
2  {
3      public bool EvalExpr()
4      {
5          Value oldValue = Thread.VolatileRead(ref value);
6          int encodedState = State;
7          int oldState = CellState.DecodeState(encodedState);
8
9          if (oldState < CellState.Computing &&
           ↪  TryTakeOwnership(encodedState)) {
10             return TryEval(oldValue);
11         } else if (oldState == CellState.Computing && Thread.Id <
           ↪  CellState.DecodeOwner(encodedState)) {
12             do {
13                 oldState = CellState.DecodeState(State);
14                 int owner = CellState.DecodeOwner(State);
15
16                 if (owner == CellState.Uptodate) {
17                     return false;
18                 } else if (TryTakeOwnership(State)) {
19                     return TryEval(oldValue);
20                 }
21             } while (Thread.Id < CellState.DecodeOwner(State));
22         }
23
24         return false;
25     }
26 }
```

Listing 5.6 – Code for parallel evaluation of a formula expression.

`oldState` and check if `oldState` is less than the `Computing` state. If it is, the cell's state must be either `Dirty` or `Enqueued` and the cell unclaimed. We attempt to claim it by calling `TryTakeOwnership` defined in listing 5.7 which uses a CAS to atomically set the cell's state to a new state that is the encoding of the current thread's ID and the `Computing` state. If we claim the cell, we call the auxiliary method `TryEval` which first evaluates the formula's expression then uses a CAS to atomically exchange the old value with the new.

The second case is given by lines 11-22. We first decode the cell state and check whether it is `Computing`. If so, we check whether the current thread has precedence over the cell's owner by comparing its ID with the ID of the owner. The do-while loop in lines 12-21 continuously tries to claim the cell by first re-reading the cell's state, as it might have changed since we last read it, and checks whether the cell has become `Uptodate`. If it has, we return false. Otherwise, we call `TryTakeOwnership` to attempt to claim the cell and evaluate it via `TryEval`. This continues

```
1  public class Formula
2  {
3      public bool TryEval(Value oldValue)
4      {
5          // Evaluate the formula's expression
6          Value result = expr.Eval();
7
8          if (Cas(ref value, result, oldValue) == oldValue) {
9              State = CellState.Uptodate;
10             return true;
11         } else {
12             return false;
13         }
14     }
15
16     public bool TryTakeOwnership(int oldState)
17     {
18         return Cas(ref state, CellState.EncodeOwner(CellState.Computing),
               ↪  oldState) == oldState;
19     }
20 }
```

Listing 5.7 – Auxiliary helper methods for parallel evaluation of a formula expression.

until we either evaluate the cell successfully, another thread computes it, or another thread with higher precedence claims the cell. We defer the exact reason for why this must be done continuously until the next section.

In the third and final case, the cell is already owned by another thread that the current thread does not have precedence over, and we simply return false in line 24 to signal that we did not evaluate the cell.

Repeated Claiming of Cells

In this section, we motivate why threads must repeatedly attempt to claim cells if they have precedence. Consider the scenario in figure 5.7a and precedence ordering $\texttt{Id}(t_j) < \texttt{Id}(t_k) < \texttt{Id}(t_i)$. Two things can happen: either thread t_j claims cell A1 before t_k which cannot claim A1 due to the precedence ordering, or t_k claims cell A1 while the CAS of t_j fails. Of course, t_j can also claim the cell from t_k but this case is not pertinent to the example. In the second outcome, t_j would simply give up altogether if it did not attempt to claim cells repeatedly. This would be problematic if there existed a cycle as depicted in figure 5.7b where t_k claims A1 while t_j fails to do so. Then t_k discovers cell C3 owned by t_j

Figure 5.7 – Two scenarios showing why threads must continuously attempt to speculatively claim ownership of cells to detect cycles properly. In the scenario on the right, threads t_j and t_k risk becoming stuck indefinitely. Support edges have been omitted for clarity.

but cannot claim it since $\texttt{Id}(t_j) < \texttt{Id}(t_k)$. Meanwhile, thread t_j is waiting indefinitely for cell A1 to finish evaluation but this will never happen and recalculation becomes stuck.

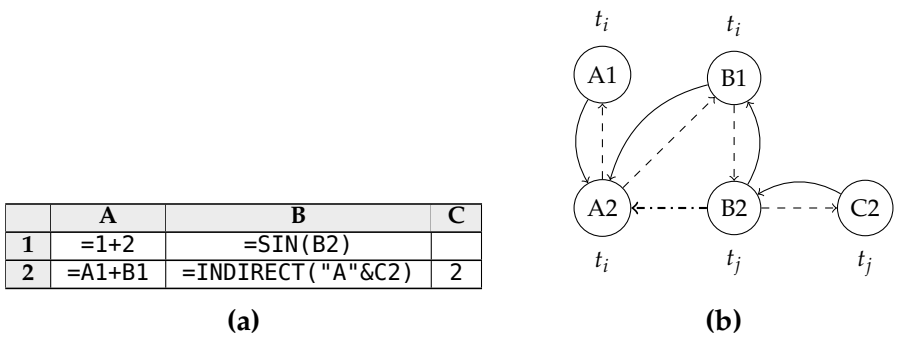
Delayed Speculative Reevaluation

Cycles are the exception, not the norm. Therefore, a thread with precedence first spins for 1 millisecond while continuously checking if the cell becomes `Uptodate` before attempting to claim it. If the cell becomes `Uptodate` while spinning, the thread does not attempt to evaluate the cell speculatively, otherwise it proceeds to repeatedly attempt to claim and evaluate the cell as before. The 1 millisecond delay was chosen since it is close to the average cell evaluation time of our benchmark spreadsheets, making sure that we do not needlessly start evaluating a cell when its result is likely available soon.

5.3.3 Dynamic Indexing Functions

In this section, we show how the algorithm handles dynamic indexing functions like `INDIRECT` and finds cyclic dependencies with the example in figure 5.8. Recall that `INDIRECT` can dynamically refer to other cells by interpreting string arguments as cell references. The dynamic dependency of B2 on A2 is highlighted with a different line style to signify that it is not an ordinary dependency.

Suppose A1 is a recalculation root and thread t_i initially claims it then evaluates cells A2 and then B1. Thread t_i does not follow a support



(a) **(b)**

Figure 5.8 – Example of how a cycle introduced by a dynamic dependency via `INDIRECT` is handled by the parallel recalculation algorithm. Owner threads are shown at each cell on the right and the dynamic dependency between B2 and A2 is denoted by a thick, dashed and dotted arrow.

edge from A2 to B2 since there is no such edge. Cell B2 is also a recalculation root as it calls the volatile `INDIRECT` function which t_j claims ownership of and evaluates its dependency on C2. Then t_j evaluates the call to `INDIRECT` and subsequently attempts to evaluate cell A2 which is owned by t_i. The situation at this point in time is depicted in figure 5.8b with cell ownership next to cells.

One of two things can happen now depending on the precedence ordering of t_i and t_j. If $\texttt{Id}(t_i) < \texttt{Id}(t_j)$, t_i will claim B2 from t_j through B1. It will evaluate B2 and its dependency A2 and discover it is already the owner of A2 thus discovering the cycle. On the other hand, if $\texttt{Id}(t_i) > \texttt{Id}(t_j)$, t_j will have precedence over t_i and claim cell A2. It will evaluate B1 and then B2, discovering itself and the cycle. Thus the implicit cycle introduced by `INDIRECT` is detected in both cases.

5.4 Thread-Local Evaluation

If a cell only supports a single cell, the main thread in listing 5.3 will still spawn a new task to compute it, even though there is no inherent parallelism to exploit. Instead, the thread could evaluate it locally to avoid additional contention on the global queue and avoid spawning a new task.

Listing 5.8 is a modified `Eval` method that now detects whether a cell only has one supported cell and calls `EvalThreadLocal` (implementa-

```
public class TaskBasedInterpreter
{
    public Value Eval(Formula cell)
    {
        switch (CellState.DecodeState(cell.State)) {
            // case CellState.Computing...

            case CellState.Dirty:
            case CellState.Enqueued:
                if (cell.EvalExpr() && UseSupportSets) {
                    if (cell.SupportCount == 1) {
                        EvalThreadLocal(cell.SupportedCells()[0]);
                    } else {
                        foreach (Cell supp in cell.SupportedCells()) {
                            Enqueue(supp);
                        }
                    }
                }
                break;

            case CellState.Uptodate:
                break;
        }

        return cell.Cached;
    }
}
```

Listing 5.8 – The code from listing 5.6 modified to support thread-local evaluation.

tion not shown) to locally evaluate the cell. Method `EvalThreadLocal` enters a while loop that continues evaluating cells locally as long as a cell only has a single supported cell. It must still successfully evaluate the cell since it may have multiple dependencies where another thread could attempt to evaluate it simultaneously. If a cell has more than one supported cell, we enqueue its support set and continue evaluation as normal.

5.5 Consistency and Correctness

In the previous sections, we explained how parallel minimal recalculation and cycle detection work. We now wish to examine if the parallel recalculation algorithm abides by the consistency requirements on recalculation of section 2.9, and examine the correctness of the cycle detection algorithm.

	A	B
1	=B1	=NOW()
2	=B1	

(a) **(b)**

Figure 5.9 – Two cells depend on the same cell containing a call to `NOW`. If both cell A1 and A2 are evaluated in parallel and both attempt to evaluate B1, their respective threads must agree upon which value B1 has evaluated to.

As illustrated in figure 5.9, two or more threads may attempt to evaluate the same cell simultaneously. In such situations, all threads must agree on the cached value of the cell. Any other threads must discard the result of their own evaluation and continue using the updated cached value. Using CAS in method `TryEval` in listing 5.7 ensures only one thread gets to update the cell's cache such that all threads agree on the value of each cell in σ. If the spreadsheet contains a cyclic dependency we cannot guarantee consistency. We let all threads continue using stale values and notify the main thread of the cycle. This approach elegantly terminates recalculation. Our algorithm thus abides by the consistency requirements except for in the presence of cyclic dependencies.

Our second concern is correctness. We do not want to produce false positives (a cycle is reported when there is none) or false negatives (a cycle is not reported when there is one). Consider any cyclic dependency of a given size with n participant threads $t_1, \ldots, t_n$. There will always be one thread t_i out of these n threads with minimal ID $\min(\{\texttt{Id}(t_j) \mid j = 1, \ldots, n\}) = \texttt{Id}(t_i)$ and consequently the highest precedence. Since thread t_i can claim all cells in the cycle, it will eventually discover it. Even if another thread t_k claims a cell that is part of the cycle and where $\texttt{Id}(t_k) < \texttt{Id}(t_i)$, t_k will then have the highest precedence and claim all cells in the cycle to discover it.

5.6 Results

To evaluate our algorithm, we ran it on six benchmark spreadsheets from LibreOffice Calc [44] and generated six synthetic spreadsheets with different topologies. The properties of all twelve spreadsheets are summarised in table 5.1. The benchmark spreadsheets and experimental set-

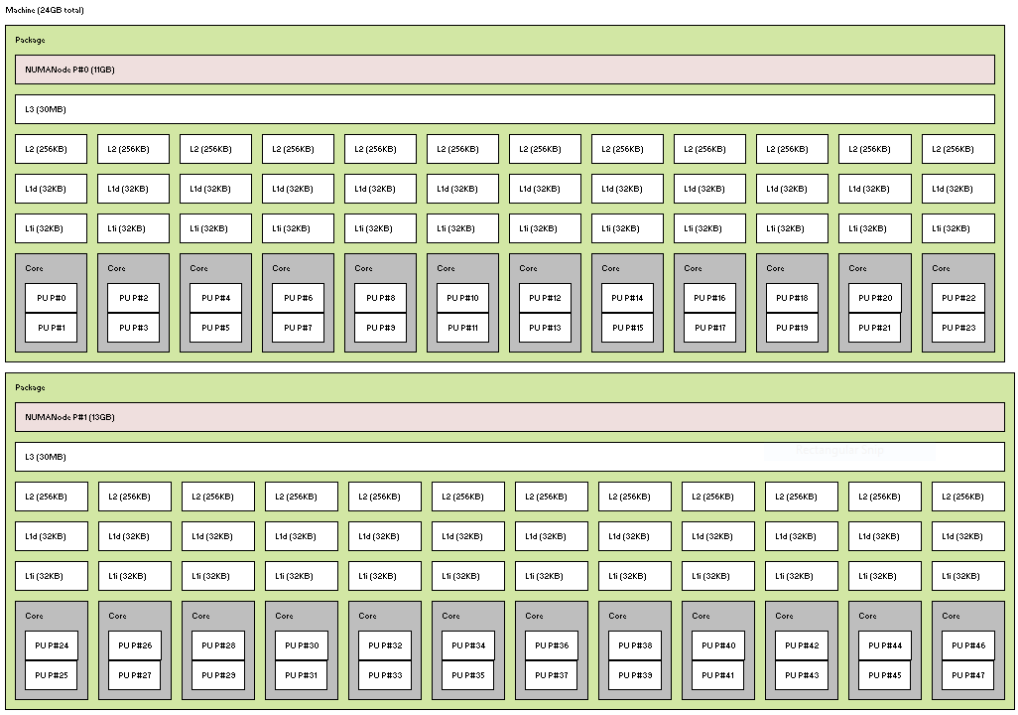


Figure 5.10 – Hardware layout of our test machine showing all cores, processor units and three-level cache hierarchy. Generated by the hwloc tool.

up described here are also used for the remaining parallel algorithms in this thesis unless otherwise specified.

5.6.1 Experimental Set-up

Our test machine was an Intel Xeon E5–2680 v3 with two separate hardware chips with 12 2.5 GHz cores each and hyperthreading (48 logical cores total), running 64-bit Windows 10 and .NET 4.7.1. Figure 5.10 shows the layout of the machine's hardware generated by the Portable Hardware Locality (`hwloc`) tool[3]. It clearly shows the two separate hardware chips with twelve cores each. Every core has a processor unit (PU), the smallest physical execution unit recognised by the tool, corresponding to the hyperthreaded capabilities of the machine. The machine has three levels of cache and the L1 cache consists of two 32 KB caches for data (L1d) and instructions (L1i).

[3] `https://www.open-mpi.org/projects/hwloc/`

Spreadsheet	Formulas	Roots	Span	Support Edges
LibreOffice Calc Spreadsheets				
building-design	108 332	18 378	4	488 351 887
energy-markets	534 507	35 198	3	287 818 610
grossprofit	135 073	15 301	3	112 612 549
ground-water	126 404	31 601	1	1 099 366 302
stock-history	226 503	23 402	3	317 049
stocks-price	812 693	10 876	3	233 376 389
Synthetic Spreadsheets				
binary-join	262 146	1	18	393 215
binary-tree	266 145	1	17	262 143
fork	300 001	1	1001	300 301
fork-join	300 002	1	1001	300 600
map	300 001	1	1	300 001
prefix	300 000	1	1100	745 009

Table 5.1 – The LibreOffice Calc and synthetic spreadsheets used for benchmarking. The **Roots** column is the number of recalculation roots selected to run a minimal recalculation that ensures that all cells in the spreadsheet are recalculated; the **Span** column is the length of the longest sequential dependency; and the **Support Edges** column denotes the total number of edges in the support graph.

For each benchmark, we initially performed 3 warm-up runs and then ran each benchmark for 5 sets of 10 iterations. We computed the average execution time for each set of 10 runs and report the average of those five averages in seconds in tables 5.2 and 5.3.

To limit the number of cores, we use the `ProcessorAffinity`[4] property to limit the logical cores on which the threads of the process can be scheduled. The property accepts a bit vector where a true bit allows the process to have its threads scheduled on the corresponding logical core. We cannot know which bit corresponds to which logical core but this is irrelevant since we only need to limit the number of logical cores, not the specific cores on which threads can be scheduled.

5.6.2 LibreOffice Calc Spreadsheets

Benchmarking on large "real-world" spreadsheets from LibreOffice Calc[5] is meant to give us insight into how well the algorithm handles realistically structured spreadsheets.

[4] `https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.process.processoraffinity?view=netframework-4.7.2`

[5] Original spreadsheets available at `https://gerrit.libreoffice.org/gitweb?p=benchmark.git;a=tree`

The spreadsheets contained some convenience macros not related to computation and functions that Funcalc does not support. To be able to benchmark the spreadsheets in Funcalc, we removed the macros and implemented unsupported functions as SDFs.

To better evaluate the performance scalability of a minimal recalculation, we found each cell that is either volatile or satisfy one of the following criteria and used them as recalculation roots.

1. The cell is a constant with a non-empty support set.
2. The cell is a formula with no dependencies e.g. `=1`.
3. The cell is part of an array formula whose expression has no dependencies.

These cells are the roots of the support graph and can collectively reach all other cells in the spreadsheet. The total number of these roots is listed in the **Roots** column of table 5.1 and can be seen as the inputs to the spreadsheet "program". While it is not so realistic to initiate minimal recalculation in this manner, it lets us examine the scalability of the algorithm and easily simulate a minimal recalculation that computes all cells in the spreadsheet.

Speed-ups *without* thread-local evaluation are plotted in figure 5.12 and *with* thread-local evaluation in figure 5.11. Table 5.2 contains the average running time for all spreadsheets across different number of cores, again with and without thread-local evaluation. Note that we also report data for 24 threads as this is the number of physical cores in our machine and anything beyond that will make use of hyperthreading.

5.6.3 Synthetic Spreadsheets

To see how the algorithm adapts to different topologies, we generated six synthetic spreadsheets with specific structures whose support graphs are shown in figure 5.13. The single striped cell serves as single a recalculation root to compute every other cell in the spreadsheet as we did with the LibreOffice Calc spreadsheets. The evaluation time for each cell was tailored to be approximately equal to the average evaluation time for cells in the LibreOffice Calc spreadsheets. In order to control the evaluation time of a cell, we defined a recursive SDF without tail-call optimisation to compute the n^{th} Fibonacci number which for $n = 20$ is close to this average evaluation time.

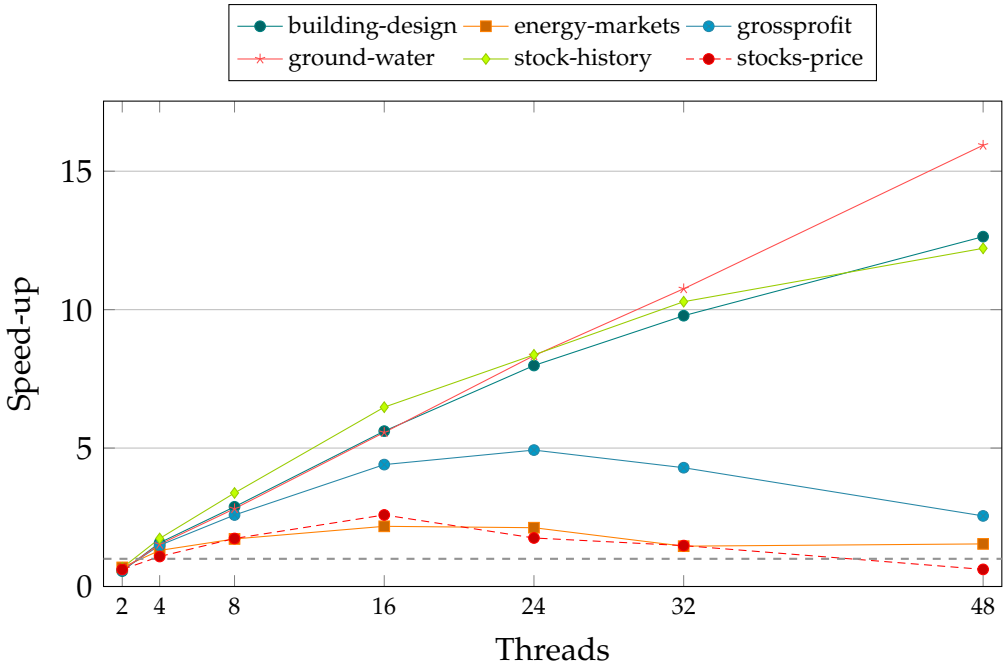


Figure 5.11 – Speed-ups over sequential performance for 50 runs of the LibreOffice Calc spreadsheets *with* thread-local evaluation. The grey, dashed line indicates the sequential baseline.

Spreadsheet	Number of Threads 1	2	4	8	16	24	32	48
Without Thread-Local Evaluation								
`building-design`	32.12	59.11	20.87	11.17	5.76	4.02	3.35	**2.49**
`energy-markets`	168.16	257.05	129.61	100.54	77.59	79.19	**69.20**	109.56
`grossprofit`	102.19	170.90	67.95	39.33	23.16	**20.74**	23.25	40.29
`ground-water`	81.26	134.21	53.32	28.84	14.85	9.75	7.69	**5.21**
`stock-history`	64.90	99.05	38.45	19.12	9.97	7.76	6.55	**5.18**
`stocks-price`	102.74	161.41	81.59	55.01	**39.91**	58.63	121.11	122.88
With Thread-Local Evaluation								
`building-design`	32.12	57.78	20.34	11.16	5.72	3.95	3.28	**2.54**
`energy-markets`	168.16	243.89	129.02	97.90	**77.37**	83.68	115.44	109.31
`grossprofit`	102.19	172.37	68.68	39.57	23.21	**20.56**	23.82	40.08
`ground-water`	81.26	136.43	53.26	28.92	14.60	9.67	7.56	**5.10**
`stock-history`	64.90	96.26	37.41	19.21	10.02	7.66	6.31	**5.31**
`stocks-price`	102.74	166.73	94.84	59.25	**39.76**	65.04	69.55	166.30

Table 5.2 – Evaluation time in seconds for different number of cores for the LibreOffice Calc spreadsheets with and without thread-local evaluation. Bolded numbers are the fastest runs per spreadsheet. The standard deviation was within ± 3.82 for all results.

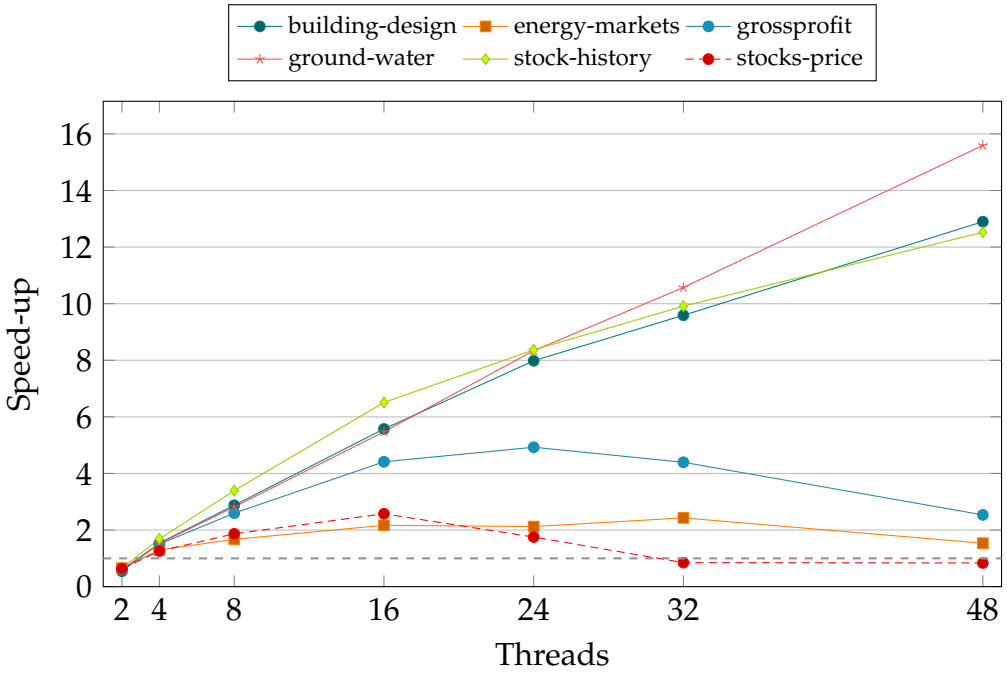
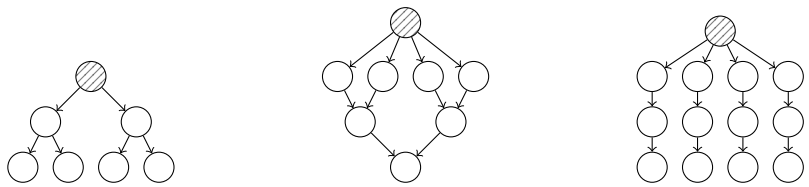
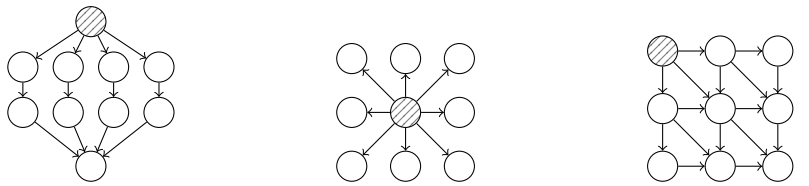


Figure 5.12 – Speed-ups over sequential performance for 50 runs of the LibreOffice Calc spreadsheets *without* thread-local evaluation. The grey, dashed line indicates the sequential baseline.



(a) Binary tree **(b)** Binary join **(c)** Fork



(d) Fork-join **(e)** Map **(f)** Prefix

Figure 5.13 – Illustrations of the underlying support graphs of the synthetic spreadsheets. Each striped node denotes the recalculation root which is used to start a minimal recalculation when benchmarking.

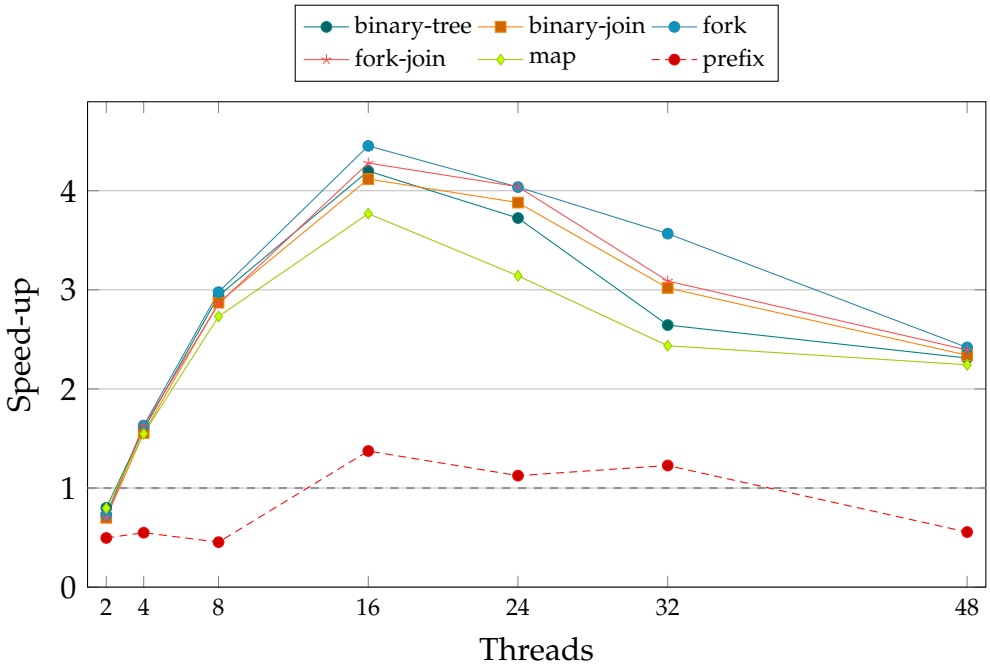


Figure 5.14 – Speed-ups over sequential performance for 50 runs of the synthetic spreadsheets *without* thread-local evaluation. The grey dashed line indicates the sequential baseline.

	Number of Threads							
Sheet	**1**	**2**	**4**	**8**	**16**	**24**	**32**	**48**
Without Thread-Local Evaluation								
`binary-join`	138.63	198.14	89.11	48.29	**33.66**	35.73	45.92	59.24
`binary-tree`	141.14	176.34	88.67	48.01	**33.62**	37.90	53.38	61.06
`fork`	160.14	219.54	98.18	53.80	**35.97**	39.67	44.89	66.22
`fork-join`	158.92	225.97	97.79	55.62	**37.14**	39.33	51.45	66.41
`map`	160.82	201.87	103.95	58.89	**42.68**	51.20	66.02	71.69
`prefix`	161.32	325.10	293.62	355.39	**117.43**	143.41	131.43	290.23
With Thread-Local Evaluation								
`binary-join`	138.63	238.03	148.18	82.87	**50.48**	64.40	64.11	116.32
`binary-tree`	141.14	181.05	87.64	48.05	**32.85**	38.92	48.89	60.96
`fork`	160.14	199.38	101.73	58.36	**39.65**	44.03	47.35	68.31
`fork-join`	158.92	236.63	120.18	68.79	**47.52**	69.68	85.65	81.38
`map`	160.82	225.24	103.69	59.51	**43.05**	50.13	65.75	71.90
`prefix`	161.32	N/A	154.16	**152.28**	157.89	236.65	206.98	463.19

Table 5.3 – Evaluation time in seconds for different number of cores for the synthetic spreadsheets with and without thread-local evaluation. Bolded numbers are the fastest runs per spreadsheet.

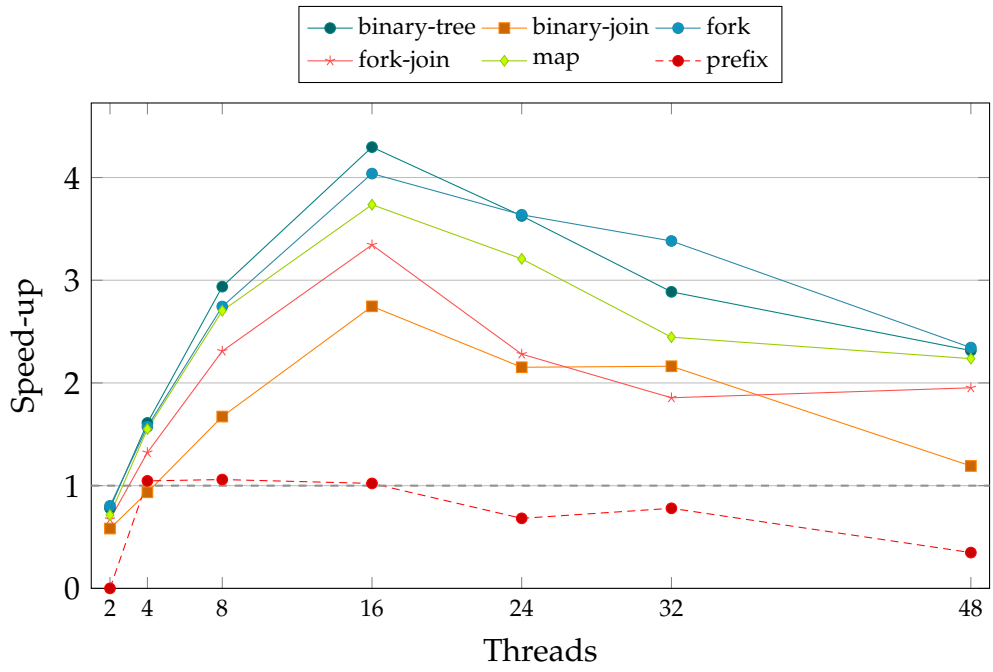


Figure 5.15 – Speed-ups over sequential performance for 50 runs of the synthetic spreadsheets *with* thread-local evaluation. The grey dashed line indicates the sequential baseline.

5.7 Discussion

There are three main observations to be made from the performance benchmarks:

Observation 1 Figures 5.11 and 5.12 show that our approach scales for the majority of the tested LibreOffice Calc spreadsheets although the performance of some drops beyond 24 cores.

The `building-design`, `ground-water` and `stock-history` spreadsheets scale, achieving a maximum 15.94-fold and 15.59-fold speed-up on 48 cores with and without thread-local evaluation respectively. On the contrary, the performance of the `energy-markets`, `grossprofit` and `stocks-price` spreadsheets drops beyond 24 cores. It is peculiar that hyperthreading benefits the performance of the first set of spreadsheets but not the others. We therefore suspect other causes as discussed next.

One explanation for the performance decline after 24 cores may simply be a lack of enough parallelism in some of the spreadsheets. This may increase contention as the number of cores increases. Another explanation is cross-chip communication between the two physically separate chips in the Intel Xeon machine. As the number of cores increase, synchronisation is more likely to happen across chips where we have to pay for expensive synchronisation when threads wait for `Computing` cells whose owners are scheduled off-chip by the TPL. The internal queue implementation of the TPL [71] is a special variant of a work-stealing queue [82, sec. 5] which allows threads with no more work in their own queues to steal work from other threads. Excessive work-stealing across chip boundaries may also impact performance although this is hard to verify. Work-stealing may increase in the poorly performing spreadsheets due to lack of parallelism as threads may migrate across cores in search of work. The structure of the three well-performing spreadsheets may also offset off-chip synchronisation e.g. if it ensures a load-balanced execution that reduces the need for work-stealing.

We also get decent speed-ups for the synthetic spreadsheets but here performance drops for more than 16 cores. Apart from the explanations we gave above, the structure and simplicity of the synthetic spreadsheets may not be enough to benefit from more than 16 cores. The `prefix` spreadsheet in particular has poor performance for all cores, barely beating sequential performance and slowing down after 16 cores. This is likely because `prefix` is the most connected spreadsheet as can be seen in the **Support Edges** column in table 5.1. This may increase synchronisation between threads that need to wait for dependencies. The algorithm is therefore not as agnostic to topology as we had initially hoped.

Observation 2 Thread-local evaluation does not appear to improve performance and leads to comparable or worse performance in most cases.

This may be due to two factors. First, thread-local evaluation is a *depth-first* traversal while normal evaluation is a *breadth-first* traversal. Thread-local evaluation makes recursive evaluation of dependencies more likely which is slower than using the global work queue. This optimisation was the main reason we included the `fork` and `fork-join`

spreadsheets where we expected to achieve better speed-ups. As can be seen from figures 5.14 to 5.15, this is not the case and it diminishes overall performance when comparing the evaluation times in table 5.3. Recursive evaluation can also lead to stack overflow errors which happened quite often for a small number of cores with the synthetic spreadsheets and thread-local evaluation. For example, it happened so often for the `prefix` spreadsheet for two cores that we could not generate any data.

Second, thread-local evaluation spawns fewer tasks and risks having more idle threads that more frequently attempt to steal work which requires synchronisation. It may also happen cross-chip.

The LibreOffice Calc spreadsheets do not benefit from thread-local evaluation either. The **Span** column in table 5.1 reveals that the longest sequential dependency is 4. This significantly reduces any performance gains from thread-local evaluation as there is no long sequential dependencies to exploit.

Observation 3 None of the spreadsheet metrics in table 5.1 are good indicators for parallel performance.

There is no apparent correlation between the metrics and the performance results. This was surprising and a deeper structural analysis is required to perhaps discover more complex causes for the observed results. The lack of insight into the performance issues reported here led us to conduct a more in-depth investigation in chapter 6.

5.7.1 Race Conditions

Unfortunately, after the algorithm's development and benchmarking, we discovered a subtle race condition which only manifests under specific conditions and in the end violates the consistency requirements stipulated in section 2.9. The race condition is best demonstrated using a minimal example such as the one in figure 5.16 and the accompanying timeline of events in figure 5.17.

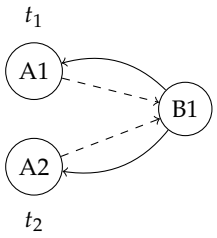
To produce the race condition, we need at least two threads, a non-deterministic function such as `RAND`, and speculative reevaluation as the cycle detection method. None of our benchmark spreadsheets contain non-deterministic functions which is why the race condition never manifested during benchmarking. Both cells A1 and A2 have been explicitly

	A	B
1	=VOLATILIZE(B1)	=RAND()
2	=VOLATILIZE(B1)	

(a)

	A	B
1	0.0	0.0
2	0.0	

(b)



(c)

	A	B
1	0.5	0.99
2	0.99	

(d)

Figure 5.16 – (a) Minimal example spreadsheet to trigger the race condition **(b)** the values of the formulas before starting a recalculation **(c)** the dependencies and support edges of the example spreadsheet **(d)** inconsistent values in cells A1 and A2 after recalculation where the race condition was triggered.

marked volatile using the `VOLATILIZE` built-in function of Funcalc. This makes sure that both cells are recalculation roots along with B1 and initially enqueued in the global work queue. If they were not recalculation roots, B1 would be computed followed by A1 and A2. As the race condition involves an inconsistency created when two threads simultaneously attempt to evaluate a cell containing a non-deterministic function, this would not trigger the race.

In figure 5.16, suppose cell B1 initially contains the value 0.0 which both A1 and A2 agree on as shown in figure 5.16b. Thread t_1 is assigned to compute A1 and t_2 is assigned to compute A2 where $\texttt{Id}(t_2) < \texttt{Id}(t_1)$. We encourage readers to also refer to listings 5.6 and 5.7.

Consider now the timeline in figure 5.17. Thread t_1 happens to act first and claims cell B1. It evaluates the call to `RAND` to 0.5 and uses a CAS to set the value of cell B1. Before it can execute the next statement to set the cell to `Uptodate` and evaluate A1, it is pre-empted and thread t_2 gets to run.

Thread t_2 sees that cell B1 has been claimed by t_1 and reads its value of 0.5 as set by t_1. Thread t_2 has precedence, waits for 1 millisecond, then claims cell B1 from t_1. Cell B1 is still not `Uptodate`. Thread t_2 then evaluates B1's formula to 0.99 but is pre-empted before it can set B1's

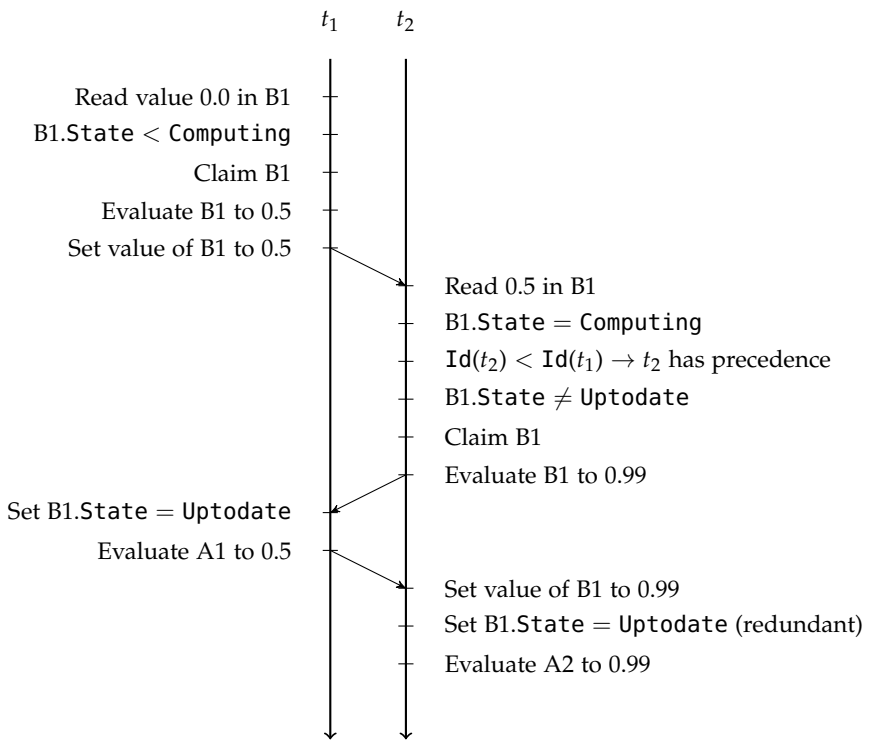


Figure 5.17 – The timeline of events of two threads t_1 and t_2 that can cause a race condition to manifest in the task-based interpreter. The events refer to the spreadsheet in figure 5.16.

value. It is important to note here that at this point t_2 has passed all checks on the cell state so it will never see another thread set the state of B1 to `Uptodate`.

Thread t_1 continues and sets cell B1 to `Uptodate` and proceeds to evaluate cell A1 that it was initially assigned to evaluate. As t_2 was pre-empted before it could set B1's value to 0.99, t_1 reads the value 0.5 in B1 and therefore sets the value of A1 to 0.5. Thread t_1 is now done.

Thread t_2 sets B1's value to 0.99 using a CAS. This succeeds because it read B1's value as 0.5, not the old value 0.0. It then redundantly sets the state of B1 to `Uptodate` and evaluates cell A2 by reading the cached value of B1, which is now 0.99 and *not* 0.5. The result of the recalculation is shown in figure 5.16d where A1 and A2 now contain different values but refer to the same cell.

This violates consistency requirement 2.2, reproduced below, which states that the formula expression $\phi(ca)$ of every cell address ca must evaluate to a value $\sigma(ca)$ that agrees with the formula expression.

$$\frac{ca \in dom(\sigma) \qquad \sigma(ca) = v}{\sigma, \alpha \vdash \texttt{ca} \Downarrow v} \text{ (e2v)}$$

Figure 5.18 – Rule (e2v) for cell reference lookup of non-blank cells. The value v looked up in the σ environment must return the same value.

$$\forall ca \in dom(\phi) \;.\; \sigma \vdash \phi(ca) \Downarrow \sigma(ca)$$

In the context of the semantic rule (e2v) for cell reference lookup of non-blank cells, reproduced in figure 5.18, σ must produce the same value for the same cell address thus the consistency requirement is violated. This produces a nonsensical result that is bound to confuse users.

Naturally, it is easier to trigger the race condition and the inconsistency with a large number of volatilized cells that all refer to a single cell calling `RAND`. We used this set-up when attempting to verify the race condition, disabled delayed speculative reevaluation, and strategically placed calls to `Thread.Sleep` in order to make it more likely to get an interleaving of events similar to the one in figure 5.17. Our example is purposefully contrived to convey the minimal conditions necessary to trigger the race.

There are thus two aspects of the task-based interpreter that we wish to improve. One is the performance issues discussed in section 5.7 and the other is the race condition presented in this section. Chapter 6 addresses the first by presenting a thorough investigation of the performance issues to uncover its causes. New insights from the investigation lead to a new algorithm, presented in chapter 7, which addresses the race condition by employing a different method for cycle detection which disallows cells from being evaluated more than once. This removes one of the conditions for the race condition.

Chapter 6

Performance Debugging

In the results of chapter 5 the `building-design`, `ground-water` and `stock-history` spreadsheets gave good speed-ups while the `energy-markets`, `grossprofit` and `stocks-price` spreadsheets yielded less impressive speed-ups, even slowdowns in some cases. This chapter details our efforts to obtain and understand the causes behind this divergence in performance. As we shall see in chapter 7, the new insights lead to an improved algorithm for parallel minimal recalculation.

We first present preliminary investigations into the performance issues in section 6.1 that look more closely at the structural layout of the LibreOffice Calc spreadsheets. In section 6.2, we turn to application profiling and obtain new insights. Finally, in section 6.3, we present two major findings using the insights from the previous section which are the foundation for the improved algorithm of the next chapter.

6.1 Preliminary Investigations

Our initial assumption was the structure of some of the spreadsheets simply did not expose enough parallelism. Therefore we measured different structural characteristics of the six spreadsheets in [2], which we also presented in table 5.1 in chapter 5 (page 68). Unfortunately, we were unable to find any correlation between these structural properties and performance scalability.

To better understand the global structure of the spreadsheets, we conducted a manual inspection of them. It quickly became apparent that the spreadsheets had one property in common: they all contained

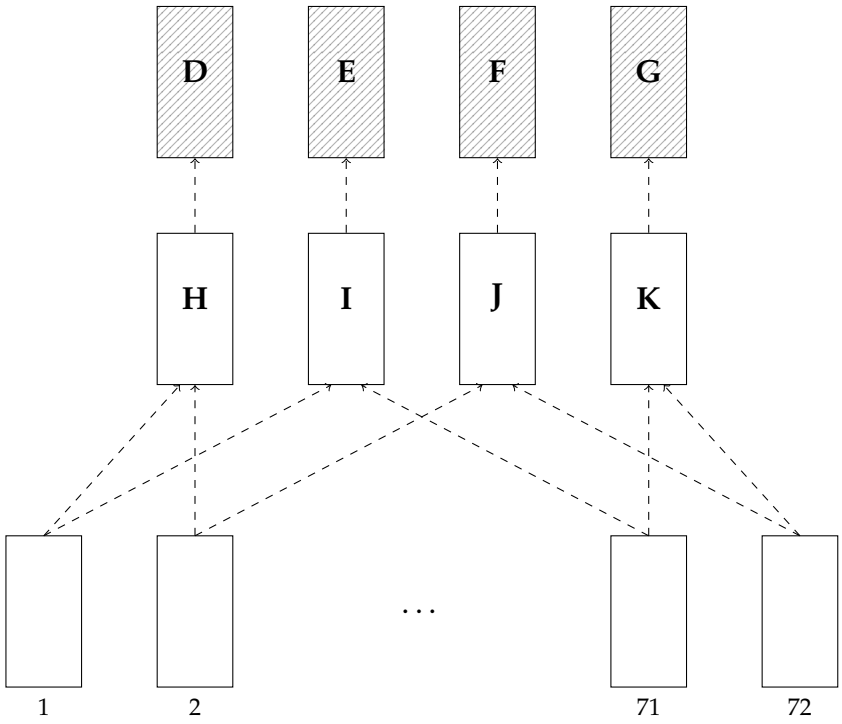


Figure 6.1 – Overview of the structure of the `energy-markets` spreadsheet. Four striped cell areas serve as a source for the remaining 76 cell arrays in the spreadsheet.

large cell arrays. This aligns with the observations of Dou et al. [24] that cell arrays are common structures in spreadsheets.

An overview of the structure of the poorly scaling `energy-markets` spreadsheet is given in figure 6.1 where dashed arrows denote dependencies between cell arrays and cell areas. The figure shows that the spreadsheet *does* in fact contain ample parallelism. Four (striped) cell areas of size 35040 containing constants in columns D, E, F and G support four cell arrays in columns H, I, J and K. These four cell arrays support the remaining 72 cell arrays in the spreadsheet which contain 5476 cells each and could be computed in parallel.

The layout of the well-performing `building-design` spreadsheet is shown in figure 6.2 and appears to exhibit some parallelism but more complicated dependencies. Six (striped) cell areas of size 18042 containing only constants in columns B, C, E, F, G and H serve as sources for the remaining cell arrays in columns D, I, J, K, L and M, also of size 18042. Cell L4 sums a subset of the constants in the cell areas of columns B and

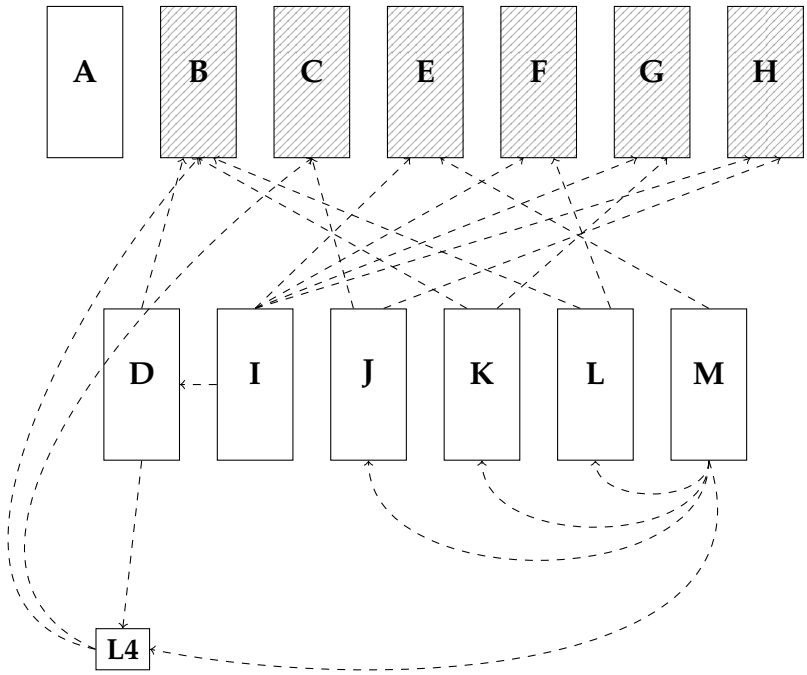


Figure 6.2 – Structural layout of the `building-design` spreadsheet. It consists of six cell areas in columns B, C, E, F, G and H with six dependent cell arrays in columns D, I, J, K, L and M. The cell in L4 contains a summation over the cell arrays in columns D and M.

C. The lone cell area of constants in column A is not referred to by any other cells since it just contains string labels.

Comparing the layouts of the two spreadsheets, we would expect the `energy-markets` spreadsheet to have comparable or better speed-ups so it appears the observed performance issues are not related to the structure of the spreadsheets.

6.2 Performance Counters

To better understand the performance issues we turned to profiling. Several tools are available for profiling .NET applications on Windows but we eventually used the Windows Performance Monitor (WPM)[1] because it was readily available on our machine and allowed us to export

[1] `https://docs.microsoft.com/en-gb/previous-versions/windows/it-pro/windows-server-2008-R2-and-2008/cc749154(v=ws.11)`

recorded data as comma-separated values (CSV) for ease of plotting. The WPM can sample several different so-called performance counters related to memory, network traffic, processors, threads and several aspects of the C# managed environment at a user-specified sampling interval. We chose to monitor the performance counters listed below.

- **% Processor Time**: The total percentage of time spent executing instructions in a non-idle thread during execution of the application. The precise sampling interval is subject to the system clock and may thus underestimate utilization.
- **% Idle Time**: The total percentage of time that all processors spend idling across the application during the sample interval. This percentage is complementary to the percentage of processor time which is why it was included.
- **% Time in GC**: The total percentage of time spent doing garbage collection since the last garbage collection cycle and reflects the last sampled value at the end of a cycle, not an overall average.

The values of these performance counters for a single run of the `energy-markets` spreadsheet for 48 cores are plotted in figure 6.3 with a sampling rate of 1 second, the lowest possible. We chose 48 cores as the values of the performance counters are more marked for a larger number of cores. It is immediately apparent that most of the time is spent doing garbage collection as the performance counter value is almost always well over 80%. Conversely, the processors only spend between 10-30% of the time presumably doing useful computation. It is worth mentioning here that processor utilisation does not equate to performance e.g. consider a spin-loop with an empty body. Although here, the poor performance combined with the low processor utilisation is a strong indication it is related to performance.

We also monitored the performance counters of the `building-design` spreadsheet, shown in figure 6.4. Since the average execution time on 48 logical cores was around 2.2 seconds and due to the WPM's coarse granularity sampling rate of 1 second, we show data across 10 consecutive runs. The time spent garbage collecting is now much lower, roughly lying between 2-10% with occasional spikes beyond 10%, 20% and 30%. Processor utilisation is now between 60-80%. The garbage collector appears to be overburdened by some part of the program in some cases

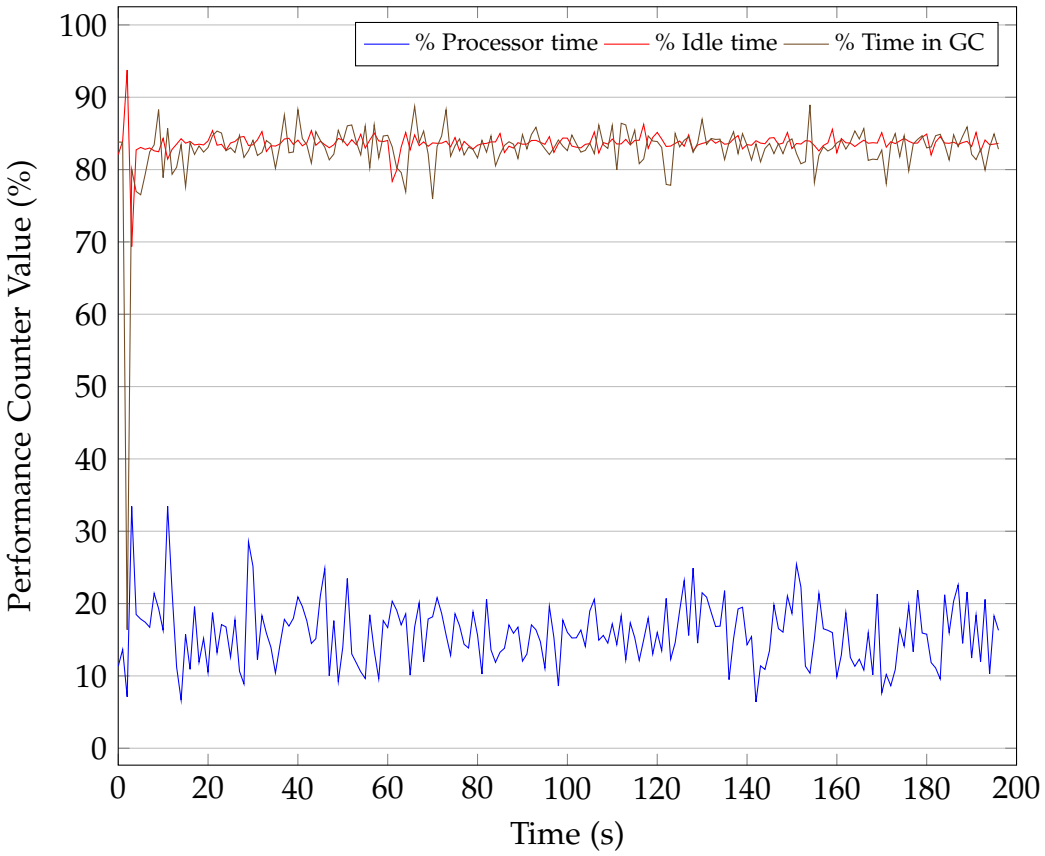


Figure 6.3 – Performance counter data for a single run of the `energy-markets` spreadsheet on 48 cores with a sampling rate of 1 second. Notice the high percentage of the time spent performing garbage collection.

resulting in poor speed-ups for some of the LibreOffice Calc spreadsheets.

6.3 Antagonistic Memory Behaviour

Following the insights of the last section, we wanted to inspect the managed heap to see what objects were being allocated and deallocated. We suspected that there would be a large amount of dead objects in the heap generations, especially generation zero and one for the younger, more short-lived objects due to many, small allocations. This suspicion arose from how garbage collection works in C# [85]. Allocations tend to be

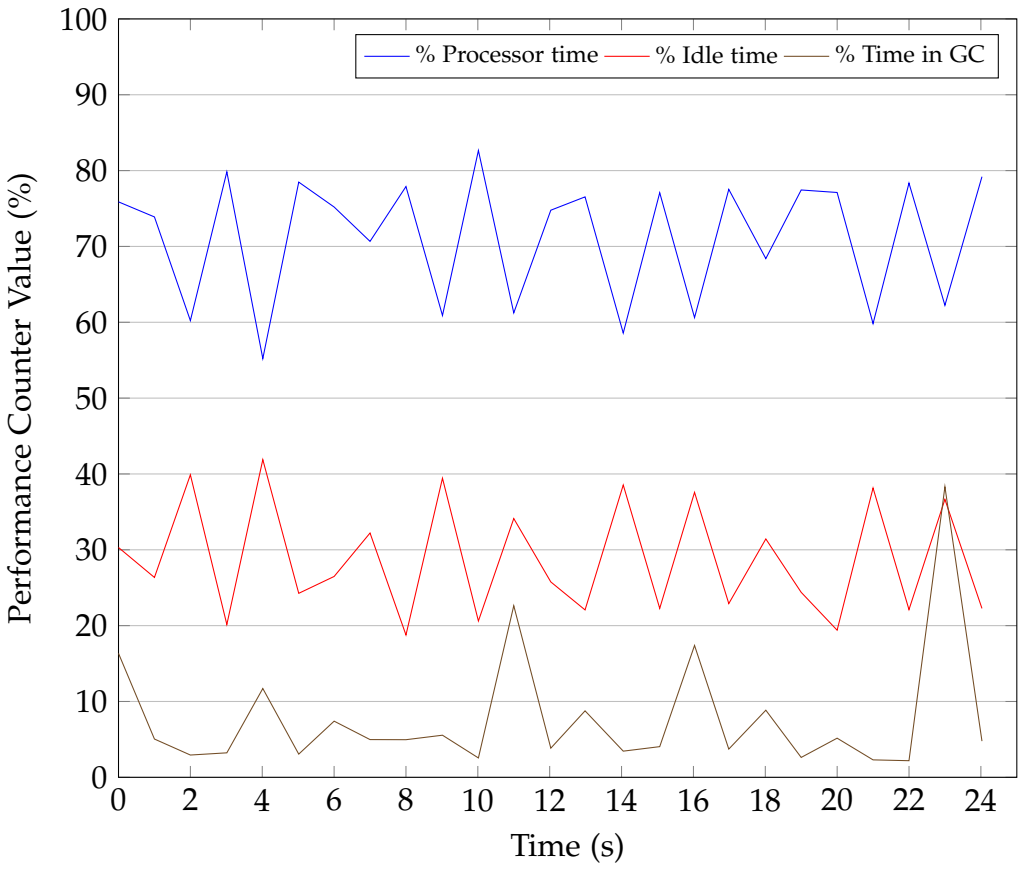


Figure 6.4 – Performance counter data for ten runs of the `building-design` spreadsheet. The sampling rate was 1 second, the lowest possible. Notice the low percentage of the time spent performing garbage collection.

relatively inexpensive but deallocation and clean-up to reclaim memory are usually not. Allocations typically involve moving a "free" pointer to point to the next free block of memory. Memory reclamation, however, uses a collection phase that finds all live objects and compacts potentially large portions of the heap to reduce fragmentation and reclaim unused memory. Live objects are found by searching from a set of root objects. The performance of the collection phase can be affected in many ways [85]. For example, stack-allocated variables are considered as roots when the garbage collector must find live objects and may increase the time taken to collect if there are many such objects on the stack.

```
0:008> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x0000019b001a1990
generation 1 starts at 0x0000019b000d2590
generation 2 starts at 0x0000019b5af81000
ephemeral segment allocation context: none
         segment             begin         allocated              size
0000019b5af80000  0000019b5af81000  0000019b6af7e8d8  0xfffd8d8(268425432)
0000019b00000000  0000019b00001000  0000019b007a1a28  0x7a0a28(7997992)
Large object heap starts at 0x0000019b6af81000
         segment             begin         allocated              size
0000019b6af80000  0000019b6af81000  0000019b6cf723d0  0x1ff13d0(33493968)
Total Size:              Size: 0x1278f6d0 (309917392) bytes.
------------------------------
GC Heap Size:            Size: 0x1278f6d0 (309917392) bytes.
```

Figure 6.5 – Output from the `eeheap` command showing information about the garbage collector heap. It shows the number of garbage collected heaps, their total size, where each generation starts and information about the allocated segments.

We used the Debugging Tools for Windows (WinDbg) program[2] and the Son of Strike (SOS) debugger extension. The WinDbg program can attach itself to a running process and let us interrupt a managed application. We can then execute SOS commands to query the state of the application's environment at the interruption point. Specifically, the SOS debugger extension allows us to examine the live and dead objects in the managed heap in each generation of the garbage collector.

The `eeheap` command outputs information about the garbage collector. An example output from the command is given in figure 6.5. It shows we have one garbage collected heap of a little over 300 MB and where each heap generation starts as well as the size and locations of each allocated segment of memory in the heap. To avoid having to figure out the generation address boundaries from the `eeheap` command output ourselves, we used to SOSEX debugger extension[3] which adds new functionality and automatically finds the correct address boundaries. The SOSEX `dumpgen` command allows us to inspect the live and dead objects of each heap generation.

Figures 6.6 to 6.8 show live and dead objects for the three generations using the SOSEX debugger extension at arbitrary points when

[2]Version 10.0.17763.1.

[3]`http://www.stevestechspot.com/SOSEXANewDebuggingExtensionForManagedCode.aspx`

```
0:008> !dumpgen 0 -stat -live
       Count      Total Size      Type
-----------------------------------------------
           4             192   Functions+<>c__DisplayClass11_0
           4             256   System.Action1[[Value]]
          14             672   Funcalc.Corecalc.Values.ArrayExplicit
          26             856   Funcalc.Corecalc.Values.ArrayValue[]
          34           1,632   Funcalc.Corecalc.Values.ArrayView
          45           2,160   Funcalc.Funcalc.FunctionValue
         127           4,680   Funcalc.Corecalc.Values.Value[]
          39         115,440   Funcalc.Corecalc.Values.Value[,]
      10,144         243,456   Funcalc.Corecalc.Values.NumberValue

10,437 objects, 369,344 bytes

0:008> !dumpgen 0 -stat -dead
       Count      Total Size      Type
-----------------------------------------------
          14             672   Funcalc.Corecalc.Values.ArrayView
          24           1,536   System.Action3[[Sheet],[Int32],[Int32]]
          42           2,016   Funcalc.Funcalc.FunctionValue
          62           2,040   Funcalc.Corecalc.Values.ArrayValue[]
          43           2,064   Functions+<>c__DisplayClass11_0
          43           2,752   System.Action1[[Value]]
          79           3,792   Funcalc.Corecalc.Values.ArrayExplicit
          87           5,568   System.Func2[[Int32],[Lazy1[[Delegate]]]]
       2,599         103,960   Funcalc.Corecalc.Workbook+<>c__DisplayClass36_1
          53         156,880   Funcalc.Corecalc.Values.Value[,]
      24,335         584,040   Funcalc.Corecalc.Values.NumberValue
      69,027       4,961,008   Funcalc.Corecalc.Values.Value[]

96,408 objects, 5,826,328 bytes
```

Figure 6.6 – Counts, sizes and types of live and dead objects in generation zero as reported by the SOSEX debugger extension.

running the `energy-markets` spreadsheet on 48 cores. Longer names have been shortened to fit on the page. The columns are the number of objects, their total size in bytes and the object type. Bear in mind that the interruption points are initiated manually and are thus arbitrary. Therefore, the SOSEX output is not a representation of the heap throughout the application's lifetime but rather a snapshot of its state at some point in time. However, we did perform numerous interruptions with WinDbg that were consistent. Three object types are particularly interesting: `Value[]`, `System.Action` and the `DisplayClass` objects. They appear both as live and dead objects in almost all generations. Other notable objects are those related to the TPL such as the `Task` object.

Other objects take up space in the generations and are related to Fun-

```
0:008> !dumpgen 1 -stat -live
       Count      Total Size      Type
-----------------------------------------------
           8             264   Funcalc.Corecalc.Values.ArrayValue[]
           6             288   Funcalc.Corecalc.Values.ArrayExplicit
          16             768   Funcalc.Funcalc.FunctionValue
          18             864   Funcalc.Corecalc.Values.ArrayView
          41           1,496   Funcalc.Corecalc.Values.Value[]
          14          41,440   Funcalc.Corecalc.Values.Value[,]
       3,891          93,384   Funcalc.Corecalc.Values.NumberValue

3,994 objects, 138,504 bytes

0:008> !dumpgen 1 -stat -dead
       Count      Total Size      Type
-----------------------------------------------
          39           1,272   Funcalc.Corecalc.Values.ArrayValue[]
          35           1,680   Funcalc.Corecalc.Values.ArrayExplicit
          52           2,496   Funcalc.Corecalc.Values.ArrayView
          70           3,360   Funcalc.Funcalc.FunctionValue
         185           6,856   Funcalc.Corecalc.Values.Value[]
          69         204,240   Funcalc.Corecalc.Values.Value[,]
      19,827         475,848   Funcalc.Corecalc.Values.NumberValue

20,277 objects, 695,752 bytes
```

Figure 6.7 – Counts, sizes and types of live and dead objects in generation one as reported by the SOSEX debugger extension.

calc's internals. The `NumberValue` object represents numbers and is created to store numbers in cells or as intermediate values passed between compiled SDFs. Two-dimensional arrays of values are represented internally as `Value[,]` objects. The `Value[,]` objects are also allocated when calling some intrinsic first- or higher-order functions that operate on arrays. The next two sections discuss the two major findings from inspection of the managed heap. We have already seen the `Formula` class for representing formula cells. The `SupportArea` object is used internally to represent large supported areas in the support graph. Section 6.3.1 focuses on the `DisplayClass` and `System.Action` objects and we return to the `Value[]` objects in section 6.3.2.

6.3.1 Allocations Associated With Tasks

TPL tasks were designed to be light-weight units of work that abstract away the low-level details of thread management. Regardless, tasks are associated with additional allocation costs.

The `Action` object denotes a C# method with `void` return type such

```
0:008> !dumpgen 2 -stat -live
       Count      Total Size      Type
-------------------------------------------------
         ...             ...   ...
       1,444         57,760   ThreadPoolWorkQueue+QueueSegment
       1,389         80,438   System.String
          37        152,440   Funcalc.Corecalc.Cells.Cell[][]
      35,086      1,122,752   Funcalc.Corecalc.Cells.QuoteCell
       1,492      3,005,408   System.Threading.IThreadPoolWorkItem[]
     148,716      3,569,184   Funcalc.Corecalc.Values.NumberValue
     140,482      4,495,424   Funcalc.Corecalc.Addressing.SupportCell
     210,244      6,727,808   Funcalc.Corecalc.Cells.NumberCell
     280,338      6,728,112   Funcalc.Corecalc.Addressing.SupportSet
       2,130      8,775,600   Funcalc.Corecalc.Cells.Cell[]
     280,338     11,213,520   List1[[SupportRange]]
     369,632     14,785,280   Funcalc.Corecalc.Workbook+<>c__DisplayClass36_1
     280,339     20,184,072   Funcalc.Corecalc.Addressing.SupportRange[]
     369,637     23,656,768   System.Action
     369,632     26,613,504   System.Threading.Tasks.Task
     534,451     29,929,256   Funcalc.Corecalc.Cells.Formula
     840,746     33,629,840   Funcalc.Corecalc.Addressing.SupportArea

3,871,794 objects, 195,118,638 bytes

0:008> !dumpgen 2 -stat -dead
       Count      Total Size      Type
-------------------------------------------------
          44          1,760   System.Threading.ThreadPoolWorkQueue+QueueSegment
         120          3,936   Funcalc.Corecalc.Values.ArrayValue[]
         251         12,048   Funcalc.Funcalc.FunctionValue
         259         12,432   Funcalc.Corecalc.Values.ArrayView
         367         17,616   Funcalc.Corecalc.Values.ArrayExplicit
         641         23,672   Funcalc.Corecalc.Values.Value[]
       1,518         85,008   System.Collections.Concurrent.VolatileBool[]
          44         91,168   System.Threading.IThreadPoolWorkItem[]
       1,518         97,152   ConcurrentQueue1+Segment[[FullCellAddr]]
       9,571        382,840   Funcalc.Corecalc.Workbook+<>c__DisplayClass36_1
       9,523        609,472   System.Action
       9,571        689,112   System.Threading.Tasks.Task
       1,518        813,648   Funcalc.Corecalc.Addressing.FullCellAddr[]
         411      1,216,560   Funcalc.Corecalc.Values.Value[,]
     147,986      3,551,664   Funcalc.Corecalc.Values.NumberValue

183,342 objects, 7,608,088 bytes
```

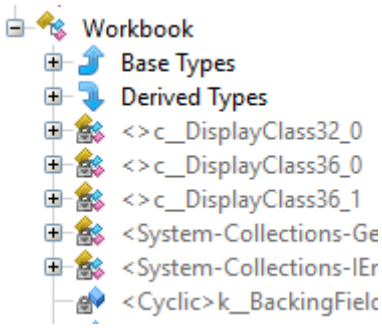
Figure 6.8 – Counts, sizes and types of live and dead objects in generation two as reported by the SOSEX debugger extension. The live object dump has been shortened to only show the objects that take up the most memory.

as the anonymous lambdas created and passed to each spawned task in the main recalculation loop of the task-based algorithm (listing 5.3 on page 54). The compiler-generated and name-mangled `DisplayClass` objects are related to closure conversion. To allow an anonymous lambda to reference local variables outside its own scope, the C# compiler generates a class to hold the referenced variables. When the C# compiler performs closure conversion of an anonymous lambda that references local variables defined outside of its scope, it generates a class to hold those referenced variables. This compiler-generated class is called a `DisplayClass` in C# terminology. We used the ILSpy .NET decompiler tool[4] to see what each `DisplayClass` is used for. Figure 6.9 shows that the compiler generates three `DisplayClass` classes for the `Workbook` class that contains the method for minimal recalculation in the Funcalc codebase. Specifically, `c__DisplayClass36_1` is generated to allow spawned tasks to reference a global exception and the `evalutingCells` variable. The global exception handles general exceptions and cyclic exceptions thrown by tasks and `evaluatingCells` is our atomic counter keeping track of the number of cells being evaluated in parallel. This particular `DisplayClass` is present in all the dumps of figures 6.6 to 6.8 and consistently present when we were inspecting the heap.

Both an `Action` and a `DisplayClass` are allocated for each spawned task and thus for each cell in the spreadsheet which quickly go out of scope again due to the relatively short average evaluation time per cell in our benchmark spreadsheets. This ultimately may overburden the garbage collector when it has to collect them again shortly after their allocation. The problem appears to worsen with an increasing number of threads and we are uncertain of the cause of this behaviour. One explanation could be that the more threads we use, the quicker short-lived objects are created, ultimately overwhelming the garbage collector faster and causing more severe, immediate heap fragmentation that prolongs the collection phase.

A closer inspection of the managed heap dumps reveal that the two object types do not take up a large portion of any of the heap generations. However, there are other factors that can affect the garbage collector. For example, the objects get promoted to generation two containing long-lived objects to our surprise. This is problematic as only a full collection can reclaim memory from generation two which involves

[4] `https://github.com/icsharpcode/ILSpy`



(a) Types and members of the `Workbook` class.

```
// Funcalc.Corecalc.Workbook.<>c__DisplayClass36_1
using ...

[CompilerGenerated]
private sealed class <>c__DisplayClass36_1
{
    public FullCellAddr next;

    public <>c__DisplayClass36_0 CS$<>8__locals1;

    internal void <RecalculatePartial>b__1()
    {
        try
        {
            next.Eval();
        }
        catch (Exception e)
        {
            CS$<>8__locals1.exception = e;
        }
        CS$<>8__locals1.<>4__this.evaluatingCells.Decrement();
    }
}
```

(b) `DisplayClass` for spawned tasks

Figure 6.9 – Screenshots from the .NET decompiler ILSpy showing that three `DisplayClass` classes are generated by the compiler for the `Workbook` class. One of them is generated to let a task reference local variables outside its scope.

halting the mutator for the duration of the reclamation [85].

6.3.2 Variadic Function Calls

Our second insight from the SOSEX debugger extension output was the `Value[]` objects which were harder to understand as the type is much more general. After thorough investigation, we found most of the objects could arise from calls to *variadic* SDFs, such as `SUM` and `MAX`, that take a variable of arguments. The generated bytecode for variadic function calls allocate an object of type `Value[]` whose size matches the number of arguments. The array is then populated with the arguments and passed to the variadic function. For example, the `PEARSON` statistical SDF used by the `energy-markets` spreadsheet, calls `AVERAGE` twice with a single cell area reference. It allocates two 1-element value arrays that can be garbage collected when the function returns.

Such single-element `Value[]` objects are allocated on the stack which are considered as roots when the garbage collector must find live objects that are not referenced by other objects [85]. This may also explain why garbage collection tends to dominate more for more threads since this increases the rate of allocation of stack-allocated variables which in turn means that the garbage collector has to consider a larger amount of roots

when reclaiming memory. For a smaller number of threads, there may be less roots to consider per garbage collection cycle, minimising the effects.

How many such calls are made in our benchmark spreadsheets? Table 6.1 shows the number of calls to built-in functions and SDFs in the LibreOffice Calc spreadsheets. It also lists the total number of variadic function calls. Simple functions for e.g. arithmetic have been omitted. Two of the poorly performing spreadsheets, `energy-markets` and `stocks-price`, call variadic functions much more often than the others. The `building-design`, `ground-water` and `stock-history` spreadsheets had better performance and call variadic functions less often. The `grossprofit` spreadsheets had poor performance but calls no variadic functions. It may be that it exposes less parallelism which appears to be the case from our manual inspections.

Function	Built-in?	Variadic Calls?	building-design	energy-markets	grossprofit	ground-water	stock-history	stocks-price
AND	Yes	-	0	4	1	0	0	0
AVERAGE	Yes	-	0	4	0	31 601	45 286	112 222
COUNT	No	No	0	0	0	0	2	0
COUNTIF	No	-	0	0	15 000	0	0	0
COUNTIFS	No	No	0	0	15 000	0	0	0
COVAR†	No	Yes	0	0	0	0	0	196 060
MAP	Yes	-	0	0	2	0	0	0
MAX	Yes	-	0	0	0	31 601	0	0
MIN	Yes	-	0	0	0	31 601	0	0
PEARSON†	Yes	Yes	0	197 136	0	0	0	0
POWER	No	No	0	140 160	0	0	0	0
PRODUCT	No	No	0	0	60 000	0	0	0
PRODUCT2	No	No	90 210	0	0	0	0	0
PRODUCT3	No	No	54 126	0	0	0	0	0
PRODUCT4	No	No	18 042	0	0	0	0	0
SLOPE†	No	Yes	0	197 136	0	0	0	0
SUM	Yes	-	54 127	0	0	0	90 574	0
SUMIFS	No	No	0	0	30 000	0	0	0
SUMPRODUCT	No	No	0	0	0	0	45 286	0
VAR	No	Yes	0	0	0	0	0	196 060
Total Variadic Calls			54 127	788 544	0	94 803	135 860	700 402

Table 6.1 – Number of function calls used in the LibreOffice Calc spreadsheets. Trivial functions such as those for arithmetic have been omitted. A superscript † denotes a function that calls a variadic function twice. The total number of variadic function calls are summed in the last row.

	Number of Threads							
Spreadsheet	**1**	**2**	**4**	**8**	**16**	**24**	**32**	**48**
Without Variadic Function Calls								
`building-design`	32.12	60.54	19.74	11.12	5.73	3.94	3.13	**2.33**
`energy-markets`	168.16	763.87	288.56	155.24	80.72	57.11	47.37	**39.60**
`grossprofit`	102.19	396.02	143.72	81.49	44.08	33.48	**28.93**	30.35
`ground-water`	81.26	134.37	52.28	28.40	14.60	9.63	7.56	**5.03**
`stock-history`	64.90	117.46	40.91	20.20	10.62	7.30	5.91	**4.41**
`stocks-price`	102.74	570.81	222.37	125.36	69.47	52.80	46.99	**44.97**
With Variadic Function Calls								
`building-design`	32.12	59.11	20.87	11.17	5.76	4.02	3.35	**2.49**
`energy-markets`	168.16	257.05	129.61	100.54	77.59	79.19	**69.20**	109.56
`grossprofit`	102.19	170.90	67.95	39.33	23.16	**20.74**	23.25	40.29
`ground-water`	81.26	134.21	53.32	28.84	14.85	9.75	7.69	**5.21**
`stock-history`	64.90	99.05	38.45	19.12	9.97	7.76	6.55	**5.18**
`stocks-price`	102.74	161.41	81.59	55.01	**39.91**	58.63	121.11	122.88

Table 6.2 – Absolute running times in seconds for each processor configuration for the LibreOffice Calc spreadsheets with and without variadic function calls. Bolded numbers are the fastest execution times per spreadsheet. Note that sequential results have been omitted.

To test that the allocation of these one-element arrays actually affects performance, we created new versions of the LibreOffice Calc spreadsheets with a custom tail-recursive definition of `AVERAGE` accepting only a single argument. Thereby eliminating the majority of variadic function calls as `AVERAGE` is called most often. The average running times are shown in table 6.2 for 20 runs total, with and without the variadic function calls to `AVERAGE`. The synthetic spreadsheets only call the simple, recursive `FIB` function for computing Fibonacci numbers and do not call any variadic functions.

Overall, the fastest evaluation time for each spreadsheet now lies at 48 cores with the exception of `grossprofit`. They are more scattered when including variadic function calls to `AVERAGE`. The standard deviation of both sets of results were very low: less than 1.30 seconds without variadic function calls and less than 3.82 seconds results with variadic function calls.

The presence or absence of the majority of variadic function calls appear to have differing effects on the performance of the spreadsheets. The three spreadsheets `building-design`, `ground-water` and `stock-history` that already performed well have not been affected significantly. The `energy-markets` spreadsheet, that called the most variadic functions, has gone from an average runtime of 69.20 seconds on 32 cores with variadic function calls down to 39.60 seconds on 48 cores

without variadic function calls, a 1.75-fold improvement. The `stocks-price` spreadsheet also called a lot of variadic functions and went up from 39.91 seconds on 8 cores with variadic function calls to 44.97 seconds on 48 cores without variadic function calls, a 0.89-fold deterioration. On the other hand, `grossprofit`, which did not call any variadic functions, went up from 20.74 seconds on 16 cores with variadic function calls to 28.93 seconds on 32 cores without variadic function calls, a 0.72-fold deterioration.

Furthermore, all spreadsheets except for `building-design` take large performance hits on 2 and 4 cores compared to the results with variadic function calls. We are unsure why this happens and the performance hit with variadic function calls is much smaller on 2 cores and completely gone on 4 cores.

Finally, our hand-optimised, tail-recursive `AVERAGE` SDF may also have an impact on performance itself so it is difficult to attribute the results in table 6.2 solely to the presence or absence of variadic function calls. Nonetheless, we monitored the same performance counters in WPM for the `energy-markets` spreadsheets on 48 cores. The results are shown in figure 6.10.

Clearly, the change has had a significant effect. The %Time in GC performance counter now lies consistently around 25-27% while the processor times floats around 70%. Apart from being a strong indication that variadic function calls can affect performance, this is also reflected in the speed-up of the spreadsheet as seen in table 6.2.

The insights gained from the profiling and investigation of performance of this section were used to develop a new and improved algorithm for parallel minimal recalculation in the next chapter that also addresses the issue of the race condition we discovered for the task-based algorithm.

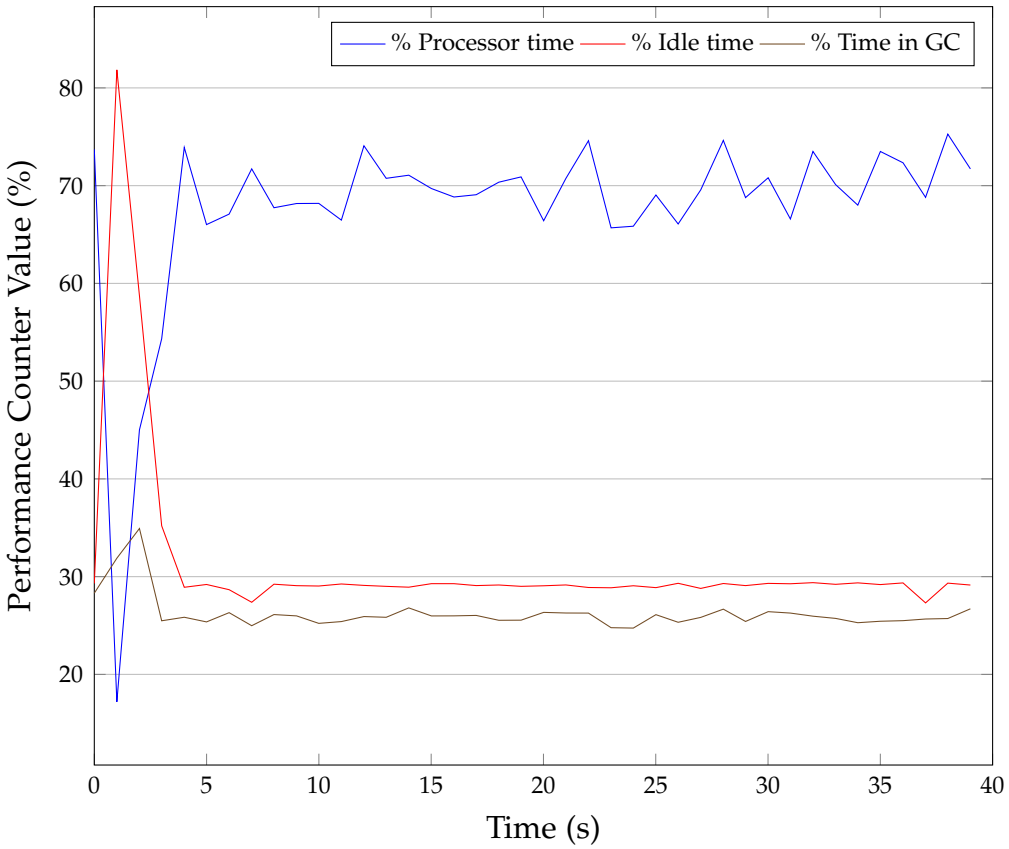


Figure 6.10 – Performance counter data for a single run of the `energy-markets` spreadsheet *without* variadic function calls on 48 cores with a sampling rate of 1 second. Notice how the percentage of time spent garbage collecting has dropped significantly compared to figure 6.3.

Chapter 7

A Thread-Based Parallel Cell Interpreter

The contents of this chapter is based on the paper "*A Parallel Spreadsheet Interpreter With Cycle Detection*" [86].

In the previous chapter we found some causes for the performance issues of the task-based parallel cell interpreter. We also found a subtle race condition in chapter 5. This chapter presents an improved parallel cell interpreter that incorporates these new insights to obtain less memory overhead and uses a new method for parallel cycle detection that avoids the data race. To eliminate the overhead associated with fine-grained per-cell task spawning, the new parallel interpreter uses *threads* instead of *tasks*. While we sacrifice the nice abstractions of the TPL, we regain full control over our threads, avoid off-chip work-stealing and hopefully alleviate garbage collection. The thread-based interpreter detects cycles in parallel without evaluating cells more than once. As a result, one of the conditions for triggering the race condition is removed. In addition, the approach is *lock-based* as opposed to the lock-free implementation of the task-based interpreter using CAS. This warrants a re-examination of the consistency requirements on recalculation and correctness in light of this new approach.

The chapter largely follows the same structure as the chapter on the task-based interpreter. It also utilises the same thread-safety measures introduced in that chapter but we highlight any differences between the two approaches as appropriate. Section 7.1 gives an overview of the thread-based recalculation process. We explain how cells are computed in parallel by each thread and present an alternative termination condition that is not dependent on the size of the shared queue. In section 7.2,

we give some informal intuition for how cycles are detected and implement cycle detection in section 7.3 along with a minor modification to the concept of cell ownership. Section 7.4 examines the consistency requirements on recalculation and argues for the correctness of cycle detection. Section 7.5 presents our results for the twelve benchmark spreadsheets which are discussed in section 7.6. Finally, some possible improvements to the algorithm are suggested in section 7.7.

7.1 Parallel Recalculation

The core idea of parallel breadth-first evaluation of cells remains unchanged. In fact, the overview of the process in figure 7.1 is very similar to the overviews of the sequential and task-based interpreters with some notable differences. We still employ a main thread to start recalculation and use the term *recalculation worker* or simply *worker thread* to denote threads responsible for evaluating cells. All worker threads are created at application start-up to match the system's number of logical processors and are assigned unique IDs $i = 1, \ldots, n$ where n is the number of threads.

The main thread marks cells dirty and enqueues the recalculation roots as before. Recalculation workers initially wait for a signal from the main thread to start recalculation. We elaborate on the meaning of "signal" later. Once signalled by the main thread, they pop cells off the queue, evaluate them and push their supported cells back onto the queue in parallel until a termination condition is met or a cyclic dependency is discovered. Each worker thread then signals the main thread that recalculation is done or has terminated prematurely due to an error. A major difference is thus that threads are themselves in charge of getting and submitting cells in the concurrent work queue `CQ`. In the task-based interpreter, dequeueing cells was the responsibility of the main thread while tasks pushed supported cells onto the queue. We first present the code for the main thread below then the code for recalculation workers and the revised termination condition in section 7.1.1.

The code executed by the main thread is shown in listing 7.1. We use the concurrent counter from chapter 5 and initialise it to zero in line 5. Cells are marked dirty and the recalculation roots enqueued as before with one exception: the `MarkDirty` method now accepts the counter as

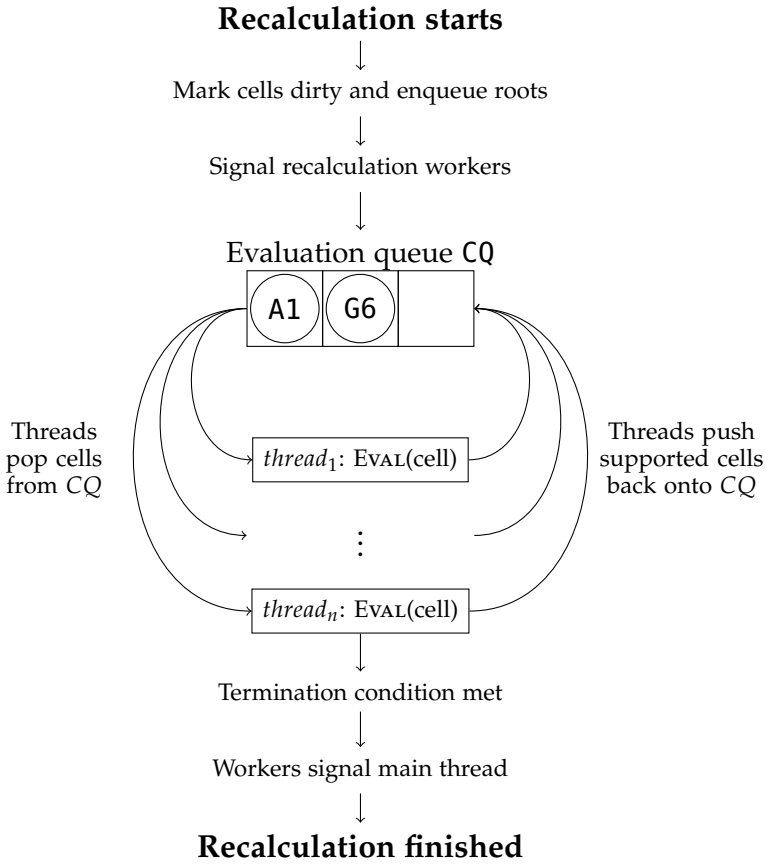


Figure 7.1 – Overview of parallel, minimal recalculation. A total of n threads wait for a signal from the main thread, then concurrently pull work off the global, shared queue and push supported cells back onto the queue in parallel until all recalculation workers are done or a termination condition is met. The threads then signal the main thread and the recalculation is finished.

an argument. It still marks `Dirty` cells reachable from the recalculation roots but also counts the number of marked cells by incrementing `counter`. This cell count is exactly the number of cells to be recalculated to complete minimal recalculation and serves as part of our new termination condition. The `Enqueue` method remains unchanged.

```
1  public class Workbook
2  {
3      public void RecalculateMinimalPar(List<Cell> roots)
4      {
5          LongAdder counter = new LongAdder(0);
6
7          foreach (Cell root in roots) {
8              MarkDirty(root, counter);
9              Enqueue(root);
10         }
11
12         Barrier barrier = new Barrier(ProcessorCount + 1);
13         Signal(barrier); // Signal worker threads to start
14         Signal(barrier); // Wait for worker threads to finish
15
16         if (CycleFound()) {
17             // Report cycle to user
18         }
19     }
20 }
```

Listing 7.1 – Code for thread-based minimal recalculation.

The main thread then allocates a synchronization primitive known as a barrier [82, chap. 17]. A barrier waits for n participant threads to signal it before letting all threads past the barrier. Until the last signal is received, threads must wait at the barrier. We create and initialise a barrier in line 12 to receive signals from as many threads as there are logical processors in the system, via the `ProcessorCount` variable, plus one for the main thread itself. Some barriers, such as the one we use, automatically reset after all participants have signalled it and can be readily reused without an explicit reset.

Worker threads patiently wait at the barrier for the main thread to signal it which happens in line 13. This is the last signal needed to allow the worker threads to go past the barrier and begin recalculation. The reset barrier is then immediately signalled again by the main thread in line 14 which blocks until the barrier is signalled by all recalculation workers. As each worker thread finishes, it signals the barrier and again waits for a new signal from the main thread to start a new recalculation

at the now reset barrier. This lets the main thread past the barrier in line 14 where it checks if a cyclic dependency was discovered in line 16. Let us now see how recalculation workers compute cells in parallel.

7.1.1 Recalculation Workers

Listing 7.2 shows the code for the `Recalculate` method of each recalculation worker. Each worker has a unique identifier, and store a reference to the barrier and the counter class. The main loop is executed as long as the `running` variable is true which is set at application start-up when worker threads are created and started. It is set to false when the application must close down. As explained previously, each worker thread waits at the barrier for the final signal in line 11. Once signalled, they enter another loop in line 13 which continues as long as there are still unevaluated cells by consulting the concurrent counter and a cycle has not been discovered. This is sufficient as a termination condition and we can disregard the size of the queue entirely. The revised termination condition is more an academic curiosity than one motivated by the performance debugging of chapter 6.

In the loop body in lines 14-18, the concurrent queue is continuously polled and cells are evaluated using the familiar `Eval` method. Cycles are still detected via `Eval` and its implementation is discussed in section 7.2. Each time a cell is successfully evaluated via `Eval`, the counter is decremented. When a worker thread is finished, it signals the barrier again in line 21.

7.2 Parallel Cycle Detection and Reachability

Our thread-based interpreter features a new cycle detection method inspired by a distributed cycle detection algorithm [87] in which nodes pass information to their neighbours in phases to convey which other nodes they can reach. After each consecutive phase, this information propagates further and further. In the presence of a cycle, some node will eventually obtain enough information to discover it. Whereas cycle detection happens in a series of phases in [87], our approach must only propagate reachability information between workers that are currently computing a cell (so its state is `Computing`) and clear this information when a cell becomes `Uptodate` since cycles only occur between cells

```
public class RecalculationWorker
{
    private int id; // Unique identifier
    private Barrier barrier;
    private LongAdder counter;
    private bool running;

    public void Recalculate()
    {
        while (running) {
            barrier.Signal(); // Wait for a signal from the main thread

            while (counter.Value > 0 && !CycleFound()) {
                Cell cell = CQ.TryDequeue();

                if (cell != null) {
                    Eval(cell);
                }
            }

            barrier.Signal(); // Signal the main thread that we are done
        }
    }
}
```

Listing 7.2 – Code executed by each recalculation worker.

whose state is `Computing`. Furthermore, we must only propagate this information through a cell's dependencies through which cycles are discovered. Propagating it through the support graph would effectively make the cell graph bidirectional and quickly lead to false positives. Another crucial property of our adaptation is to disallow cells from being evaluated more than once which removes one of the conditions for the race condition in the task-based interpreter. In section 7.4, we discuss this in more depth.

7.2.1 Reachability Matrix

Let us first establish some intuition for how we detect cycles in parallel before delving into the details of the actual implementation. We use a *reachability matrix* R, as given by definition 1, to record which threads can be reached by other threads. In the definition, we use the `Id` function to retrieve the unique identifier of a thread.

Definition 1. Let R be a binary reachability matrix indexed by i and j. If $R[i, j] = 1$, we say that thread t_k with $\texttt{Id}(t_k) = i$ can be reached by

thread t_m with $\texttt{Id}(t_m) = j$, and t_k is currently computing a cell whose state is thus `Computing`, otherwise t_m cannot reach t_k and $R[i, j] = 0$.

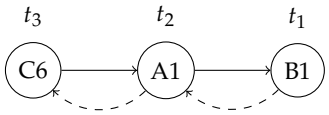
In the second part of definition 1 where t_m cannot reach t_k, we do not require that worker t_k is not currently computing a cell even if it can be reached by a worker t_m. This is because t_m may simply not yet have discovered t_k.

Consider the illustrative example in figure 7.2a where thread t_1 is evaluating cell B1 which depends on A1 owned by t_2. In turn, A1 depends on C6 owned by t_3. For the purposes of the example, $\texttt{Id}(t_i) = i$ for all three threads and we assume that we have a similar encoding scheme for cell ownership as we did for the task-based recalculation algorithm.

When t_1 wants to evaluate A1, it sees its state is currently `Computing` and it is owned by t_2 by examining the ownership bits of its cell state. Thread t_1 therefore sets $R[2, 1] = 1$, as shown in figure 7.2b, to record it can reach t_2. Likewise, t_2 can record that it can reach t_3 in a single step along the dependency graph, while t_3 itself has no dependencies. The reachability matrix is now as shown in figure 7.2c.

Once a thread is waiting for a cell to finish computing, it starts to propagate reachability information by querying R to discover which other workers it can reach through the owner of its immediately adjacent cell dependency. We denote this owner as t_{adj} in the context of a worker thread e.g. $t_{adj} = t_2$ for t_1. For example, t_1 can query if it can reach t_3 through t_2 by examining if $R[3, 2] = 1$, which is true, and since $R[2, 1] = 1$, t_1 can reach t_3 through t_2. It can thus set $R[3, 1] = 1$ to record that it can transitively reach t_3 through t_2. We call these *transitive queries* and they happen in round robin while a thread is waiting for t_{adj} to finish evaluating a cell. We revisit the round robin querying scheme when we discuss the implementation. Transitive queries correspond to examining the column of t_{adj} in R, e.g. the second column in figure 7.2b tells us which workers t_2 can reach etc. The only exception is we do not perform transitive queries of t_{adj} since we already know it is reachable as a prerequisite to performing transitive queries in the first place. For example, t_1 already knows that it can reach t_2 in figure 7.2c. This implies that worker threads can also query if they can reach themselves.

Returning to the example, t_3 eventually finishes evaluation of C6 and must set the cell's state to `Uptodate`, clearing the ownership bits, and *reset* its row in R because it is no longer reachable for the purposes of cy-



(a)

	t_1	t_2	t_3
t_1	0	0	0
t_2	1	0	0
t_3	0	0	0

(b)

	t_1	t_2	t_3
t_1	0	0	0
t_2	1	0	0
t_3	0	1	0

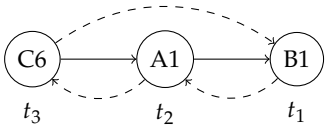
(c)

Figure 7.2 – Example of three worker threads evaluating three cells C6, A1 and B1, and updating the reachability matrix to reflect what other worker threads each one can reach. For example, in **(b)**, t_1 has discovered it can reach t_2 as $R[2,1] = 1$.

cle detection. The state change signals t_2 that C6 has been evaluated and the update to R signals that t_3 is no longer reachable, at least not until it starts computing another cell. Both actions must happen atomically to ensure correctness as discussed later. Once t_2 is done, it performs the same actions to signal t_1 that cell A1 is `Uptodate`. Finally, the value of B1 can be computed by t_1.

How does this let us detect cycles? According to definition 1, any worker thread t_i with $\texttt{Id}(t_i) = j$ that sees $R[j,j] = 1$ in the diagonal of R (highlighted in figure 7.2c) can reach itself directly or transitively, and we have found a cyclic reference.

Suppose now that cell C6 depended on cell B1 as in figure 7.3a. First, all threads discover the owner t_{adj} of their single cell dependency and subsequently what worker threads they can reach in two steps in the dependency graph via transitive queries. The reachability matrix at this point in time is shown in figure 7.3b and is just one possible intermediate outcome. Any one of t_1, t_2 or t_3 can now query if they can transitively reach themselves and discover the cycle. Suppose t_2 asks if it can reach itself transitively through its dependency C6 owned by t_3. Since $R[3,2] = R[1,3] = R[2,1] = 1$, which corresponds to the path $t_2 \rightarrow t_3 \rightarrow t_1 \rightarrow t_2$ in the dependency graph, t_2 can reach itself. It sets $R[2,2] = 1$ and discovers the cycle. The key idea is that the cycle could be discovered at this time because the entries $R[3,2]$, $R[1,3]$ and $R[2,1]$ were set by propagating information ultimately giving t_2 enough information to discover it.



(a)

	t_1	t_2	t_3
t_1	0	1	1
t_2	1	0	1
t_3	1	1	0

(b)

Figure 7.3 – A possible outcome of R in **(b)** if the example scenario in figure 7.2a contained a cyclic dependency. The difference is that cell C6 now depends on B1. Every worker has recorded that they can reach the owner of their immediate cell dependency and the owner of the cell dependency in two steps along the dependency graph.

7.3 Implementation of Parallel Cycle Detection

Now that readers have some informal intuition about how cycles are detected, we can discuss the actual implementation. First, we revisit and revise the concept of cell ownership in section 7.3.1 originally introduced in chapter 5 and explain why this is necessary. We discuss how we represent the reachability matrix in practice in section 7.3.2. Finally, we present the implementation for the `Eval` method and the implementation of the cycle detection algorithm in section 7.3.3.

7.3.1 Modified Encoding of Cell Ownership

As the reachability matrix R is indexed by worker IDs from 1 to n, we need to make a minor modification to the encoding scheme for cell ownership and consequently also to the encoding and decoding methods that we introduced earlier in chapter 5. Cell ownership still serves the same purpose as before: ensure that worker threads can claim ownership of cells and allow them to decipher the owner of a cell whose state is `Computing`.

As before, we need only two bits to represent all four cell states. Ownership is encoded in the remaining bits by flipping the $(j+2)^{th}$ bit of a worker t_i with $\texttt{Id}(t_i) = j$. An example is shown in figure 7.4 for $\texttt{Id}(t_i) = 9$ using a 32-bit integer where the state bits are encoded in the two least significant bits as before. The ownership bits are still only relevant for the `Computing` state and are zero for the three other states.

The encoding and decoding methods are shown in listing 7.3. In order to encode all 48 logical threads in our Xeon machine, we need

at least $48 + 2 = 50$ bits for ownership and cell state. Therefore, the `CellState` class now uses a `long` or 64-bit integer in listing 7.3 to represent cell states. Only the `EncodeOwner` method has changed to accept and encode a worker thread ID instead of using the thread ID assigned by the runtime.

We also need methods for handling the worker bit which are implemented in the `RecalculationWorker` class. The `OwnerBit` method is used to set a worker's bit in the reachability matrix R by left-shifting it to its appropriate position and add the `Computing` state bits using a bitwise `OR`. To retrieve and decode an ownership bit, we first use the `DecodeOwner` method from the `CellState` class (not shown) to get the ownership bits and pass them to `OwnerFromBit`. It uses the logarithmic function to base 2 to convert the flipped bit's position to a worker ID. For example, a worker thread t_i with $\texttt{Id}(t_i) = 9$ would be encoded as in figure 7.4. The ownership bits in decimal is 256 which would be then be decoded as $log_2(256) = 8 + 1 = 9$.

```
1  public static class CellState
2  {
3      // ...
4
5      private const int Shift = 2;
6
7      public static long EncodeOwner(int owner)
8      {
9          return (1L << (Shift + owner - 1)) | Computing;
10     }
11 }
12
13 public class RecalculationWorker
14 {
15     // ...
16
17     private static long OwnerToBit(int owner)
18     {
19         return 1L << (owner - 1);
20     }
21
22     private static int OwnerFromBit(long encodedOwner)
23     {
24         return (int)Math.Log(encodedOwner, 2) + 1;
25     }
26 }
```

Listing 7.3 – Methods for encoding and decoding state and ownership bits.

$$\underbrace{00000000000000000000100000000}_{\text{Ownership bits (256)}}\ \underbrace{10}_{\text{State bits (2)}} = 1026$$

Figure 7.4 – Encoding ownership in a 32-bit integer. The 30 most significant bits encode the single ownership bit and the 2 least significant bits encode the `Computing` state (2).

7.3.2 Reachability Matrix Representation

In practice, R is represented by an array of integers such that each row of the matrix corresponds to a separate integer and each column c is represented by the c^{th} bit of each integer. This gives us a light-weight data structure with a one-off allocation cost for a given number of workers. Updating R becomes a simple matter of bit manipulation and a row in R is cleared by setting the integer to zero. We encode bits in R starting from the least significant bit so position $R[1,2]$ represents the second bit of the first integer in the array as indices are one-based. In the implementation however, we naturally use zero-based array indices. We discuss this choice of representation and alternative ways to represent R in section 7.7. Since R is shared between all worker threads, reads and writes to each row of R is protected by its own lock stored in a separate array. Using locks enables us to set a cell's state to `Uptodate` and manipulate an entry in R in the same critical section which is necessary for correctness as we discuss in section 7.4. Listing 7.4 lists the methods for updating R as informally described in the previous section. We explain each method in the order they are given in the code listing.

To clear reachability for a worker, `ClearReachability` on line 6 first acquires the lock for the worker's row in R, sets the cell's state to `Uptodate`, assigns zero to the worker's row in R to clear reachability, then releases the lock. Consequently, any worker thread that acquires the same lock afterwards will see that the cell is `Uptodate` and not update R.

Method `TransitiveQuery` on line 14 performs transitive queries in round robin when a worker is waiting for a dependency to complete. It accepts the ID of the dependency's owner (t_{adj}) and the formula cell being computed by the current worker thread. The `NextQueryId` (code omitted) returns identifiers for all other workers in round robin except for the `owner` ID, i.e the owner t_{adj}. Recall that we do not perform transitive queries of t_{adj} since we trivially know that it is reachable. Transitive

queries are done in round robin to reduce the duration for which the lock is held. Alternatively, we could acquire the lock and perform all transitive queries in one go but this increases the time a lock is held. Since we assume that cycles are the exception and not the norm, we opt for the round robin approach.

Returning to the implementation of `TransitiveQuery`, we acquire the lock for the worker of the transitive query and retrieve its row in R. In line 24, we check if `owner` can still reach the worker that we transitively query, i.e. if its bit is still set: $R[tcQueryId, owner] = 1$. If it cannot reach it, we cannot propagate any information in this case so we release the lock and return. Otherwise, we check if the state of the formula cell became `Uptodate` before we acquired the lock (in which case we must *not* update R). If not, we can still reach the worker and set the bit of the transitively reachable worker in R using `OwnerBit`. Finally, we release the lock and return the ID of t_{adj} of the transitive query for use in the next method.

```
1  public class RecalculationWorker
2  {
3      private static long[] R;       // The reachability matrix
4      private static object[] locks; // Locks for each entry in R
5
6      void ClearReachability(Formula cell)
7      {
8          lock (locks[this.id - 1]) {
9              cell.State = CellState.Uptodate;
10             R[this.id - 1] = 0L;
11         }
12     }
13
14     int TransitiveQuery(int owner, Formula cell)
15     {
16         // Retrieve next id for transitive query
17         int tcQueryId = NextQueryId(owner);
18
19         lock (locks[tcQueryId - 1]) {
20             long tcReachability = R[tcQueryId - 1];
21
22             // Check if owner (t_adj) can still reach the worker thread
23             // of the transitive query (tcQueryId)
24             if ((tcReachability & OwnerToBit(owner)) != 0L) {
25                 if (cell.State != CellState.Uptodate) {
26                     R[tcQueryId - 1] |= OwnerToBit(this.id);
27                 }
28             }
29         }
30
31         return tcQueryId;
32     }
```

```
33
34      bool UpdateTransitiveReachability(int owner, Formula cell)
35      {
36          int tcQueryId = TransitiveQuery(owner, cell);
37
38          if (tcQueryId == this.id) {
39              // We checked if we could reach ourselves transitively
40              lock (locks[this.id - 1]) {
41                  return (R[this.id - 1] & OwnerToBit(this.id)) != 0L;
42              }
43          }
44
45          return false;
46      }
47
48      void UpdateAdjacentOwner(int owner, Formula formula)
49      {
50          lock (locks[owner - 1]) {
51              if (formula.State != CellState.Uptodate) {
52                  R[owner - 1] |= OwnerToBit(this.id);
53              }
54          }
55      }
56  }
```

Listing 7.4 – Methods in the `RecalculationWorker` class for updating the reachability matrix R.

Method `UpdateTransitiveReachability` uses `TransitiveQuery` to update R. If the ID of the transitive query `tcQueryId` happened to be a query of the current recalculation worker, we check if it can now reach itself, i.e. if $R[id, id] = 1$. If `tcQueryId` is different from the current work thread's ID, there is no reason to perform this check and we return false. If we can reach ourselves, we return true to indicate that we have found a cyclic reference.

The last method `UpdateAdjacentOwner` is used to initially record that a worker thread can reach the owner of its immediate adjacent dependency (t_{adj}) and works similarly to `TransitiveQuery`.

7.3.3 Cell Evaluation and Cycle Detection

We can now present the code for the missing piece in our thread-based recalculation algorithm for minimal recalculation: the `Eval` method for evaluating individual cells and detecting cycles which uses the methods

introduced in the previous section. The code for the method is given in listing 7.5 and we explain it in terms of the four possible cell states.

```
1  public class RecalculationWorker
2  {
3      private bool followingDependency;
4
5      public Value Eval(Formula cell)
6      {
7          long encodedState = cell.State;
8
9          switch (CellState.DecodeState(encodedState)) {
10             case CellState.Computing:
11                 if (followingDependency) {
12                     // We are evaluating via dependencies
13                     int owner = OwnerFrom-
                        ↪  Bit(CellState.DecodeOwner(encodedState));
14
15                     if (this.id == owner) {
16                         SetCycleFound(cell);
17                     } else {
18                         UpdateAdjacentOwner(owner, cell);
19
20                         // Spin waiting for the dependency to complete
21                         while (CellState.DecodeState(cell.State) !=
                            ↪  CellState.Uptodate) {
22                             if (UpdateTransitiveReachability(owner,
                                ↪  cell)) {
23                                 SetCycleFound(cell);
24                             }
25                         }
26                     }
27                 }
28                 break;
29
30             case CellState.Dirty:
31             case CellState.Enqueued:
32                 if (cell.EvalExpr(this.id)) {
33                     cellCount.Decrement();
34                     ClearReachability(cell);
35
36                     if (UseSupportSets) {
37                         foreach (Cell supp in cell.SupportedCells()) {
38                             Enqueue(supp);
39                         }
40                     }
41                 }
42                 break;
43
44             case CellState.Uptodate:
45                 break;
46         }
47
48         return cell.Cached;
```

```
49      }
50  }
```

Listing 7.5 – Code for evaluating a cell as done by each recalculation worker.

If the state is `Computing`, we check if we are following a dependency. The worker-local variable `followingDependency` declared in line 3 is set to true whenever a worker follows a dependency via a cell reference or cell area reference. We decode the owner from the encoded state and check if we have discovered ourselves in line 15. If so, we call `SetCycleFound` (code not shown) which atomically sets a shared flag and calls `ClearReachability` to set the cell's state to `Uptodate`. The shared flag is returned by `CycleFound` which is part of the termination condition of listing 7.2 in section 7.1.1. As in the task-based algorithm, setting the cell's state to `Uptodate` enables any pending workers to finish computing with a stale value and then immediately find that a cycle has been discovered when they call `CycleFound` to terminate recalculation.

If we do not own the dependency, we first record the owner of the dependency in R via `UpdateAdjacentOwner`, then enter a loop in lines 21-25 where we continuously perform transitive queries via `UpdateTransitiveReachability` until the cell becomes `Uptodate` or a cycle is discovered. If we are not following a dependency but evaluating a cell via the support graph, we simply wait for the cell to become `Uptodate` in line 48.

If the cell state is either `Dirty` or `Enqueued`, we attempt to claim ownership of it and evaluate the formula's expression via `EvalExpr` in line 32. Recall that `EvalExpr` recursively evaluates a cell's dependencies through which we detect cycles and propagate reachability information. It sets the `followingDependency` variable to true if we evaluate any dependencies and back to false upon return. This was not necessary in our task-based interpreter since speculative reevaluation allowed threads to proceed and detect cycles through dependencies as in the sequential interpreter. No information needed to be propagated in a certain direction. Here, each cell is evaluated only once so the implementation of `EvalExpr` can simply claim the cell via a CAS, evaluate the cell's expression and set its value. It accepts a worker ID to encode in the ownership bits when attempting to claim the cell. If we successfully claim it,

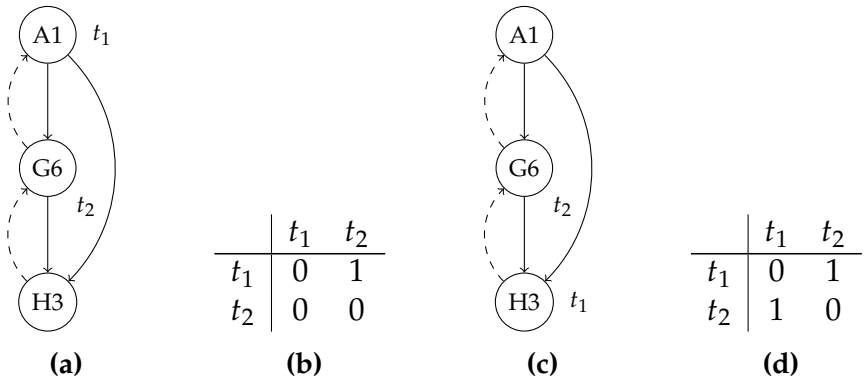


Figure 7.5 – A scenario where an incorrect implementation of the algorithm might report a false positive cyclic dependency. The initial scenario and reachability matrix is shown on the left in **(a)** and **(b)**. The scenario just before detection of the false positive is shown on the right in **(d)** and **(c)** when t_1 has evaluated cell A1.

we decrement the counter since the cell has now been evaluated, clear the worker's row in R via `ClearReachability`, and enqueue the cell's supported cells onto the evaluation queue.

7.4 Correctness

Having explained the algorithm for cycle detection, we now argue for its correctness and explain how it avoids the race condition we discovered in the task-based algorithm.

Part of the correctness argument uses *linearisation points* [82, chap. 3.5]: the point of a method where it instantaneously appears to take effect between its invocation and return. For lock-based implementations, a critical section is a linearisation point as its effects become visible to other threads when the lock is released.

A crucial property of the cycle detection algorithm is that it must not be possible for a worker t_i to finish evaluating a cell, clear its reachability and then continue recalculating while some other thread t_j believes that it can still reach t_i. In this case, t_i may later rediscover itself incorrectly. In other words, a worker thread must *only* be reachable if it is computing a cell.

Figure 7.5 depicts such a situation where $\texttt{Id}(t_i) = i$. In figure 7.5a, t_1 is computing cell A1 while t_2 is computing cell G6 which depends

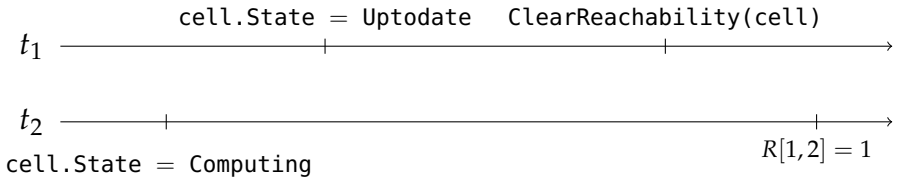


Figure 7.6 – A naive lock-free implementation of the algorithm using CAS might miss updates which in turn might lead to false positive cycles being detected.

on cell A1 so $R[1,2] = 1$ in figure 7.5b. In figure 7.5c, t_1 has evaluated A1 and begins evaluating cell H3 which is supported by A1. It depends on G6 so $R[2,1] = 1$ as shown in figure 7.5d. Thread t_1 assumes that its reachability has been cleared but in reality t_2 still incorrectly believes that it can reach t_1. When t_1 evaluates H3 it can discover itself through cell G6 which would cause a false positive cycle to be reported.

This could happen if we used separate CAS intrinsics for setting the cell state to `Uptodate` and clearing R. Consequently, we create a small timing window between the two actions since they each happen atomically but not together in a single atomic operation. We saw a similar timing window in the race condition of the task-based algorithm. Consider the series of actions in the timeline of figure 7.6 where thread t_2 is computing a cell that depends on a cell owned by t_1. Assume that every action is done using CAS, $\texttt{Id}(t_i) = i$, and $R[1,2] = 0$.

Initially, the t_1's cell is in the `Computing` state which is observed by t_2. Before t_2 can update R, t_1 sets the cell state to `Uptodate` and clears its entry in R. When t_2 subsequently wants to update R, it operates under the assumption that the cell state is still `Computing` and incorrectly records that it can reach t_1. This can cause t_1 to discover itself as exemplified in figure 7.5. This issue is the primary reason for using locks as now both actions happen atomically together. Later in section 7.7, we discuss ways in which one could go about implementing a correct lock-free version of the cycle detection algorithm.

Recall that the definition of the reachability matrix states a worker is either computing a cell and can be reached by other workers, or its reachability is zero in R. By using locks, we make certain that workers first successfully acquire the lock before setting both the cell state and clearing a row in R in a single linearisable action. There is no intermediate state where the cell is `Uptodate` but other threads still believe

the owner of the cell is reachable. Consequently, both actions become visible to other threads at the same time. Any worker acquiring the same lock afterwards will see that the cell has become `Uptodate` and not incorrectly update R.

Our new cycle detection method disallows speculative reevaluations so each cell is evaluated only once. This removes one of the required conditions for the race condition of the task-based algorithm to manifest, also removing the risk of the race condition here. When claiming a cell, a CAS on the cell's state ensures that only one thread gets to evaluate the cell. Also since we never acquire more than one lock at any given time we never risk deadlock.

7.5 Results

To evaluate the thread-based algorithm, we ran it on the same twelve benchmark spreadsheets used for the task-based interpreter using the same experimental set-up. However, we use the LibreOffice Calc spreadsheets without any variadic function call to `AVERAGE` that we developed in chapter 6.

Figure 7.7 plots the speed-ups for the LibreOffice Calc spreadsheets without variadic function calls and figure 7.8 plots the speed-ups for the synthetic spreadsheets. Table 7.1 lists the running time in seconds for the runs of all twelve spreadsheets.

Naturally, we were also interested in examining the impact on performance of using locks for cycle detection. To measure this impact, we again turned to the WPM. We tracked the % Processor Time, % Idle Time and % Time in GC performance counters like before but added two new performance counters that are listed and described below. By tracking the % Time in GC we can verify if our efforts to remove the performance issues discussed in chapter 6 had an impact on garbage collection.

- **Contention Rate / Sec**: The number of contentions per second i.e. the number of failed acquisitions of a managed lock. The WPM scales this counter by a factor of 0.1.

- **Total # of Contentions**: The total number of contentions since the application started. In the performance counter plots of figure 7.9 this value is initially positive since we still perform three warm-up runs. The WPM also scales this counter by a factor of 0.1.

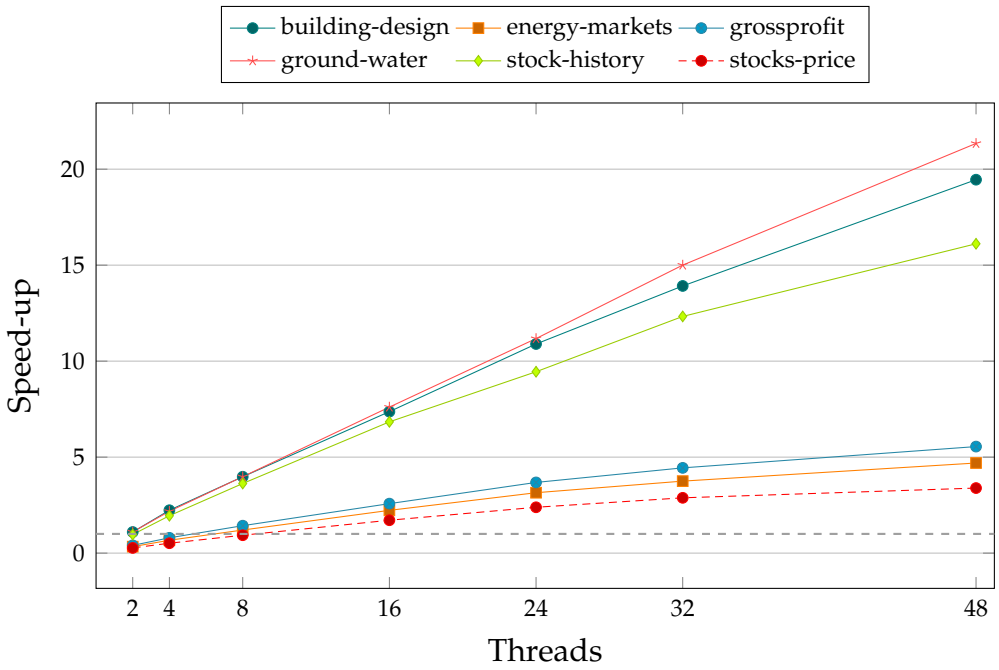


Figure 7.7 – Speed-ups over sequential performance for 20 runs of the LibreOffice Calc spreadsheets. The grey, dashed line indicates the sequential baseline.

The results for single runs of the six LibreOffice Calc spreadsheets are shown in figure 7.9 using 48 threads to attempt to maximize contention. Like before, we used the smallest possible sampling rate of 1 second which is why the results for the spreadsheets with good scalability have very few data points.

7.6 Discussion

We make four key observations from figures 7.7 and 7.8.

Observation 1 We get overall positive speed-ups for the LibreOffice Calc spreadsheets where the best performance is at 48 cores. All speed-ups beat those of the task-based interpreter. The performance of the `energy-markets`, `grossprofit` and `stocks-price` spreadsheets does not worsen after 24 cores.

The `building-design`, `ground-water` and `stock-history` spreadsheets still scale much better than the `energy-markets`, `grossprofit`

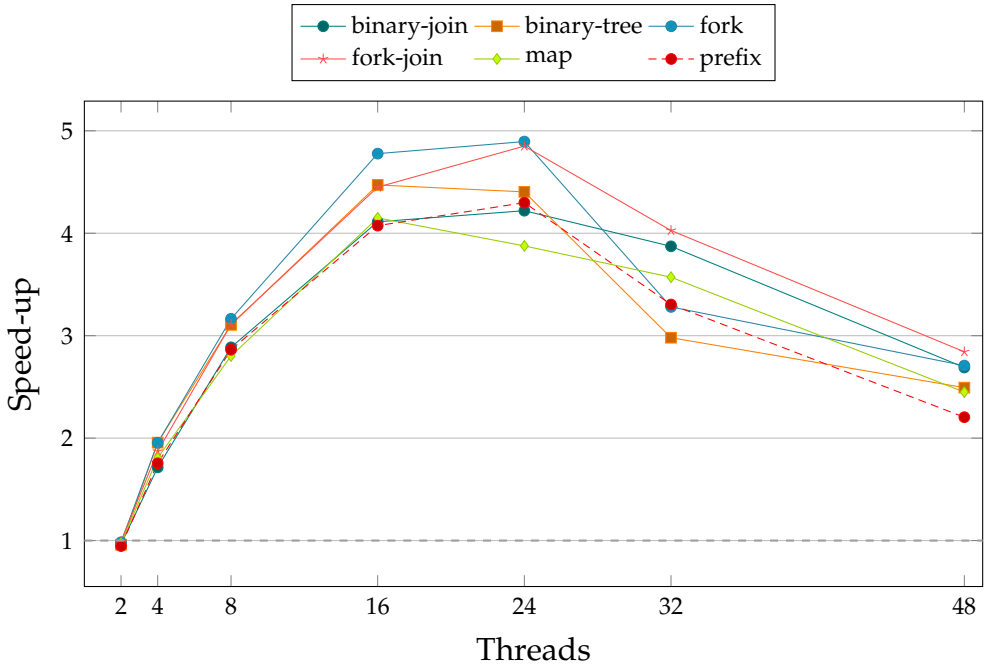


Figure 7.8 – Speed-ups over sequential performance for 20 runs of the synthetic spreadsheets. The grey, dashed line indicates the sequential baseline.

	Number of Threads							
Spreadsheet	**1**	**2**	**4**	**8**	**16**	**24**	**32**	**48**
LibreOffice Calc Spreadsheets								
`building-design`	32.12	29.24	14.38	8.07	4.36	2.95	2.31	**1.65**
`energy-markets`	168.16	499.37	250.93	139.84	75.50	53.45	44.85	**35.83**
`grossprofit`	102.19	253.61	128.23	71.52	39.65	27.75	23.01	**18.42**
`ground-water`	81.26	73.68	37.28	20.41	10.68	7.27	5.42	**3.81**
`stock-history`	64.90	67.01	33.33	17.90	9.49	6.87	5.26	**4.03**
`stocks-price`	102.74	380.00	199.45	110.24	59.98	43.03	35.67	**30.32**
Synthetic Spreadsheets								
`binary-join`	138.63	143.62	80.87	48.03	33.71	**32.85**	35.80	51.54
`binary-tree`	141.14	147.12	72.16	45.44	**31.57**	32.05	47.36	56.62
`fork`	160.14	162.86	81.94	50.60	33.52	**32.71**	48.81	59.14
`fork-join`	158.92	160.25	84.94	51.08	35.71	**32.76**	39.46	55.91
`map`	160.82	165.40	88.97	57.40	**38.76**	41.49	45.04	65.66
`prefix`	161.32	170.49	91.94	56.36	39.59	**37.53**	48.82	73.19

Table 7.1 – Evaluation time in seconds for different number of cores. Bolded numbers are the fastest runs per spreadsheet. The standard deviation was within ± 1.03 for all results except for `energy-markets` on 32 cores with a standard deviation of ± 11.18.

and `stocks-price` spreadsheets. We now achieve a maximum 21.34-fold speed-up for the `ground-water` spreadsheet.

Fortunately, our efforts to remove some of performance issues of the task-based interpreter appear to have paid off as the three poorly performing spreadsheets now give better speed-ups between 3.39- and 5.55-fold and do not exhibit the drops in performance beyond 24 cores that we previously observed. We are still not fully satisfied with their performance and give suggestions for additional improvements in section 7.7.

Observation 2 The `energy-markets`, `grossprofit` and `stocks-price` spreadsheets take a big performance hit at fewer than 8 cores as can be seen in table 7.1.

This is slightly surprising since one would usually expect such behaviour at 2 or perhaps 4 cores because as the core count increases, the benefit of multi-threading tends to outweighs its overhead. This behaviour could also observed for the task-based interpreter but only for 2 cores and we are unsure what the exact cause is. It is less prominent for the well-performing spreadsheets where we get a speed-up on 2 cores except for the `stock-history` spreadsheet whereas we did not get any speed-up on 2 cores in the task-based interpreter.

Observation 3 Hyperthreading seems to benefit the LibreOffice Calc spreadsheets as performance continues to increase beyond 24 cores, whereas it drops for the synthetic spreadsheets beyond 24 cores.

This was also the case for the task-based interpreter with the exception that performance dropped for more than 16 cores for the synthetic spreadsheets. Our main assumption is that locking in the cycle detection algorithm causes threads to be de-scheduled more often, giving hyperthreading room to run another thread on the same core.

Hyperthreading does not benefit the synthetic spreadsheets as performance drops beyond 24 cores. The simple dependencies in the synthetic spreadsheets may not be enough for threads to yield control via locking and reap the benefits of hyperthreading. Even though the average evaluation time per cell was tailored to match the LibreOffice Calc spreadsheets, the distribution of evaluation times which may also affect locking as no cell takes longer to compute than any other cell. The LibreOffice Calc spreadsheets also contain more complex functions causing

threads to wait long enough for hyperthreading to have a positive impact in general.

Observation 4 The thread-based interpreter appears to be largely agnostic to different spreadsheet topologies.

A major difference in the performance of the task-based interpreter for the synthetic spreadsheets is that the `prefix` spreadsheet now has comparable performance to the remaining five synthetic spreadsheets, even slightly higher speed-ups in some cases compared to e.g. the `map` spreadsheet which has significantly fewer dependencies than `prefix`. This would suggest that the thread-based interpreter is more resilient to changes in topology than the task-based interpreter.

The use of locking in the cycle detection method is likely not an issue. If it was, we would expect the performance of more connected spreadsheets such as `prefix` to suffer as multiple threads attempt to acquire the lock for the same worker much more often than in less connected spreadsheets. Spreadsheets with cells that support many other cells and take much longer to evaluate may suffer much more as the owners of supported cells all attempt to acquire the same lock.

We may attribute this behaviour to thread "jumping". When a thread is done evaluating a cell, it pulls another cell off the global work queue. This new cell may lie in an entirely different position in the spreadsheet than the cell the worker thread just evaluated. We may view the thread as having "jumped" from one position in the spreadsheet to another. This may minimise the risk of threads acquiring the same locks on the same entries in the reachability matrix as opposed to threads being more localised during evaluation. In a highly connected spreadsheet like `prefix`, this may be very beneficial.

Why did we not observe this with the task-based algorithm? One explanation may be that CAS operations are not without downsides. Many repeated retries on a shared resource may be expensive if there is high contention for the resource. This may happen very often in highly connected spreadsheets like `prefix` whereas the effect may be less severe using locking where threads are descheduled and hyperthreading can schedule another thread to run.

7.6.1 Lock Contention and Garbage Collection

We now examine the performance counter plots of figure 7.9 to better understand the impact of lock contention on performance. We make three key observations from the data.

Observation 1 Lock contention has a clear impact on processor utilization across all spreadsheets and may thus affect performance.

This is especially visible in the `ground-water` spreadsheet where processor utilization drops from close to 100% to around 55% when lock contention spikes after approximately 2 seconds. Similar spikes and drops in processor utilisation are observable in the other plots. Not surprisingly, idle time increases when processor utilization drops, however time spent garbage collecting also increases perhaps in response to the inactivity of the application threads if they are de-scheduled when locking giving the garbage collector time to do its job.

Observation 2 Even though lock contention appears to affect performance, we still achieve good speed-ups on most spreadsheets.

Hyperthreading may help mitigate drops in processor utilization by scheduling another thread on a core, at least for the LibreOffice Calc spreadsheets. We would postulate that the poor speed-ups on the LibreOffice Calc spreadsheets are perhaps caused or dominated by another factor than lock contention. We observed the same division of performance in the task-based interpreter which suggests another performance artefact.

Observation 3 Memory overhead is significantly reduced and consequently less garbage collection takes place.

Much to our delight, our efforts to reduce the negative effects of garbage collection appear to have paid off. When examining the plots in figure 7.9, the value of the % Time in GC performance counter is consistently below 30% except in cases where lock contention spikes. As we saw in chapter 6 on performance debugging, the % Time in GC value for the `energy-markets` spreadsheet now lies between 20%-25% whereas it laid well above 80% before, and the % Processor Time performance counter now lies between 70%-80% instead of the 10%-30% range.

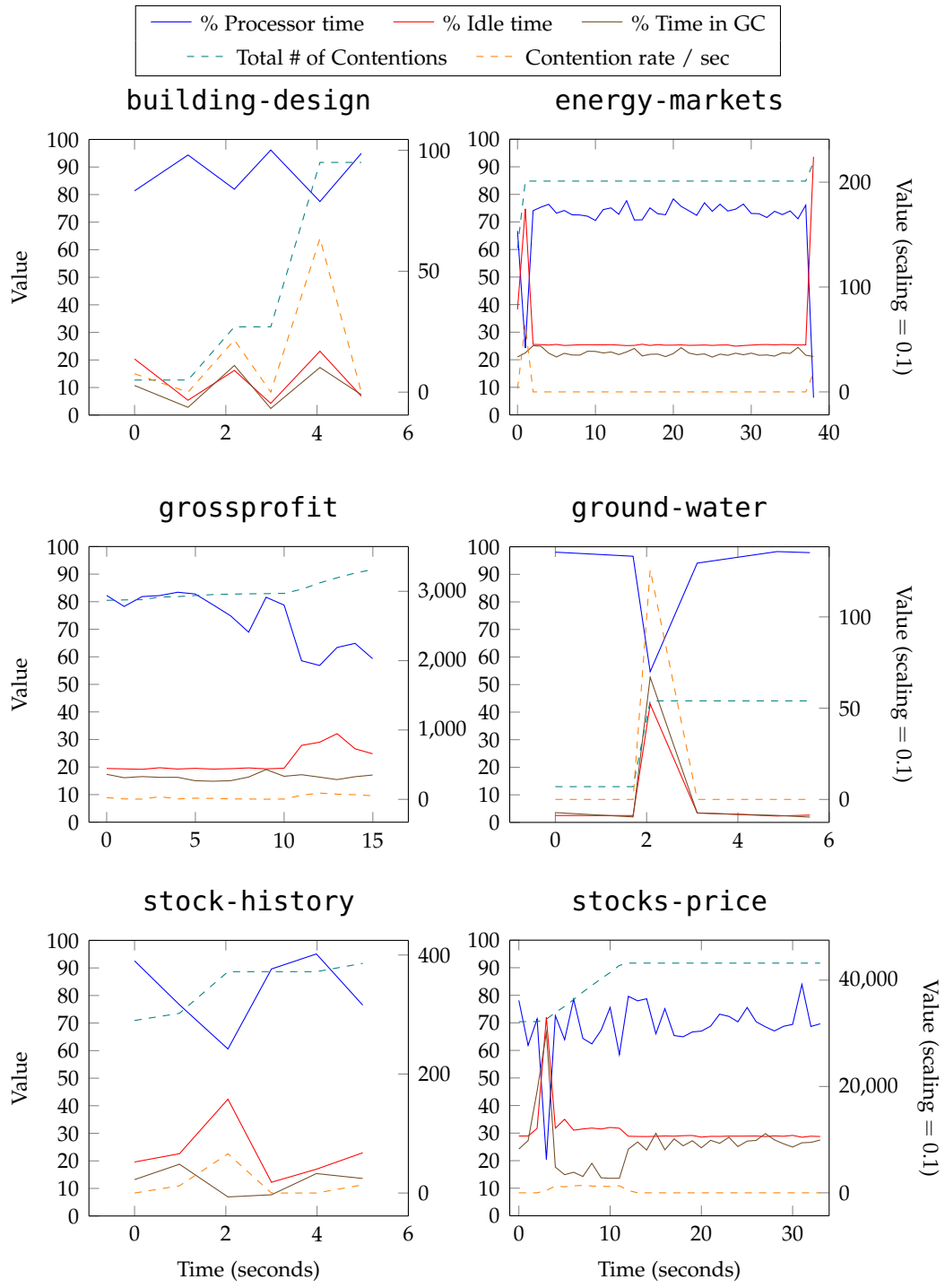


Figure 7.9 – Different performance counter measurements for single runs of the six LibreOffice Calc spreadsheets with 48 threads to maximise contention. The lock contention rate per second and the total number of contentions are automatically scaled by a factor of 0.1 by the WPM hence the second y-axis on the right.

While processor utilisation does not necessarily translate to performance, it is safe to assume that the higher rates of processor utilisation translate to performance judging when considering the speed-ups we achieve.

7.7 Future Work

Before concluding we discuss some possible improvements to the algorithm to be implemented in future work.

7.7.1 Representation of the Reachability Matrix

We chose to represent the reachability matrix R as an array of integers which limits the number of threads to the number of bits in the largest integral type supported by the CAS intrinsic of a given system, save for two bits for the cell state. On our 64-bit Intel Xeon machine, we can thus only scale to 62 threads. However, since our algorithm chiefly targets commodity hardware, the current scalability limitations should suffice for today's systems. Hopefully as hardware supports an increasing number of threads, the size of integral types and CAS instructions will evolve alongside it.

Despite limiting scalability, our choice of representation for R incurs only a one-time allocation cost and has no memory overhead associated with updating the data structure. Still, one might consider other data structures to circumvent scalability limitations such as a per-worker concurrent set data structure [82, chap. 13] protected by the same set of locks. This would likely incur slightly more memory traffic due to allocations and deallocations in the data structure but would allow the systems to scale.

7.7.2 Lock-Free Implementation of Cycle Detection

We chose to implement parallel cycle detection using locks to ensure correctness of the algorithm but locks are not without downsides. In general, they do not tend to scale well, can deadlock, and can starve threads. If a thread crashes when holding a lock, the system can stall indefinitely. Fortunately, deadlock is not an issue here due to the absence of nested lock acquisition.

However, we believe a lock-free implementation is possible but it would need more sophisticated hardware instructions than CAS since a naive implementation quickly breaks down as we discussed in section 7.4.

One solution might be the load-link (LL) and store-conditional (SC) pair of instructions. The first one loads a memory address and the second stores a value at the memory address. The SC instruction also checks if the memory location was modified since the last LL instruction regardless of which value is stored there. Therefore, this pair of instructions do not suffer from the ABA problem [82] as CAS does. Brown et al. [88] provide the load-link extended (LLX) and store-conditional extended (SCX) synchronization primitives that can be implemented in software using CAS.

Another solution would be to use instructions such as double-length compare-and-swap (DCAS) or multi-word compare-and-swap (MCAS) where we can atomically set both cell state and an entry in R atomically. Unfortunately, all these instructions are not as widely available as the CAS instruction.

Lastly, we could use a regular CAS but reserve some bits for a special tag value [89]. Whenever we update a value using CAS, we increment the tag value as well. Therefore, we can distinguish between two CAS updates with the same value since their tags will differ. This solution is theoretically unsound [90] however, as it would need unbounded tags to avoid issues with tags wrapping around but this issue may prove to have an infinitesimal risk of occurring in practice.

7.7.3 Thread-local Queues

In our task-based approach, we developed a variant of the algorithm that used thread-local evaluation to avoid spawning tasks for sequential chains of supported cells. If there is only a single supported cell, it would be wasteful to enqueue it in the global queue and spawn a separate task for it, as we could instead just evaluate it ourselves.

We could implement a similar approach for the thread-based algorithm where we instead always enqueue a single supported cell in a thread-local queue regardless of the number of supported cells to minimise contention on global queue and greedily keep some work for ourselves. If there is only a single supported cell the implementation would act identically to the task-based implementation as the single supported

cell would be enqueued in the thread-local queue and no cells are enqueued in the global queue.

While this idea may prove useful for the thread-based approach for spreadsheet such as the `fork` or `fork-join` spreadsheets, it may also counteract the effects of thread "jumping" that we mentioned in section 7.5 as an explanation for the good performance of the highly connected `prefix` spreadsheet. Thread-local queues would localise a thread in an area in the spreadsheet to a higher degree which could increase lock acquisitions of the same locks.

We could also take this idea to the extreme by keeping as much work as possible local and only submit a small amount of work to the global queue to promote thread-local evaluation of cells.

Part II

Static Partitioning of Spreadsheets

Chapter 8

Big-Step Cost Semantics

The contents of this chapter are taken from the technical report "*Concrete and Abstract Cost Semantics for Spreadsheets*" [6] that is joint work with Thomas Bøgholm, Peter Sestoft, Bent Thomsen and Lone Leth Thomsen.

The parallel cell interpreters presented in the last couple of chapters dynamically exploit local parallelism during cell evaluation in minimal recalculation. In the second part of the dissertation, we instead globally partition the spreadsheet into load-balanced groups of cells that can be efficiently run on shared-memory multicore processors. We hope to achieve a better work distribution that can achieve higher speed-ups at the cost of performing a static analysis to partition the spreadsheet. This approach thus focuses on full recalculation where all cells must be recalculated.

A prerequisite to any static partitioning algorithm is a good approximate cost model to help load-balance work. One part of our cost model is a big-step cost semantics obtained from direct extension of the big-step semantics we introduced in section 2.8 for a small spreadsheet language. Furthermore, we extend the cost semantics to encompass features of Funcalc such as higher-order functions. The cost semantics is used to estimate the work of each cell in the static partitioning algorithm introduced in chapter 9.

Augmenting operational semantics with costs is not a new idea. For example, Blelloch et al. [91] defined an operational semantics for the NESL programming language that included both work and depth to model the amount of parallelism. Given a directed acyclic graph (DAG) of a program's sequential control dependences, work is the number of

nodes in the DAG i.e. the total amount of work in the program and depth is the largest depth of the DAG i.e. the longest sequential dependency in the program. However, to the best of our knowledge, this is the first thorough specification of a cost semantics for spreadsheets which has other uses apart from partitioning. We discuss some of these alternative uses in section 8.6.

The rest of the chapter is as follows. We introduce the big-step cost semantics in section 8.1. The cost semantics is used to estimate the total cost of both minimal and full recalculation in section 8.2. We extend the consistency requirements of recalculation in section 8.3 to account for costs. We implemented the cost semantics in Funcalc to estimate the cost of cells and discuss interesting aspects of this implementation in section 8.4. In section 8.5, we evaluate the cost of the LibreOffice Calc spreadsheets and a subset of spreadsheets from the EUSES corpus [92]. Lastly, we discuss uses of the natural cost semantics beyond static partitioning in section 8.6.

8.1 Concrete Big-Step Cost Semantics

In this section, we extend the grammar, semantic sets and partial functions of our small spreadsheet language to accommodate SDFs supported by Funcalc in section 8.1.1. In section 8.1.2, we extend the semantics from section 2.8 to include costs. Section 8.1.3, we present rules for evaluation of first-order intrinsic functions, followed by higher-order intrinsic functions in section 8.1.4.

8.1.1 Extending Grammar and Environments

Figure 8.1 shows the extended syntax for the spreadsheet language. There are four new syntactical constructs. The first construct calls a SDF sdf with expressions $e_1, \ldots, e_n$. The second uses the `CLOSURE` intrinsic function to create a closure or function value that is a partial application of an existing n-ary SDF and a set of expressions $e_1, \ldots, e_k$ where $k \leq n$. Closures can be further partially applied, again using the `CLOSURE` function, by providing an existing closure e_0 and further arguments $e_1, \ldots, e_n$ similar to currying in functional programming. The last construct uses the `APPLY` function to fully apply a closure denoted by expression e_0 and any remaining n arguments $e_1, \ldots, e_n$ necessary to apply the closure.

e	::=	`n`	number constant
	|	`ca`	cell reference, e.g. B2 or G$6
	|	`IF`(e_1,e_2,e_3)	conditional expression
	|	`RAND`$()$	volatile function
	|	`F`$(e_1,\ldots,e_n)$	call to built-in function, e.g. `=SUM(A1, B2)`
	|	$\texttt{ca}_1 : \texttt{ca}_2$	cell area reference, e.g. A1:B2 or G$6:C1
	|	$ae[i,j]$	array formula component
	|	$sdf(e_1,\ldots,e_n)$	call to a sheet-defined function
	|	`CLOSURE`$(sdf,e_1,\ldots,e_k)$	closure creation
	|	`CLOSURE`$(e_0,e_1,\ldots,e_n)$	closure partial application
	|	`APPLY`$(e_0,e_1,\ldots,e_n)$	closure full application
ae	::=	e	array expression

Figure 8.1 – Extended syntax for the small spreadsheet formula language with calls to SDFs as well as and closure creation and application [6].

As an example, let us create a closure of the `INDEX` function. It accepts an array or cell area reference and a one-based row and column index. The closure will always index an unspecified row in the second column of its argument as follows. The `NA` function, which returns a `#NA` "not available" error value, is used to mark late-bound arguments.

```
=CLOSURE("INDEX", NA(), NA(), 2)
```

We could further partially apply the closure to always index at the first row and second column, still leaving the target array or cell area reference unspecified.

```
=CLOSURE("INDEX", NA(), 1, 2)
```

The partially applied closure could then be used to index an array using `APPLY`.

```
=APPLY(CLOSURE("INDEX", NA(), 1, 2), A1:B2)
```

The closure application would retrieve the value in cell B1. The semantic sets and partial functions will likewise need to be extended to accommodate SDFs and closures as shown in figure 8.2.

$$
\begin{array}{llll}
n & \in & Number & = \{ \text{ proper numbers } \} \\
 & & Error & = \{ \texttt{\#DIV/0!}, \texttt{\#CYCLE!} \} \\
av & \in & ArrVal & = \{ Av(w, h, [[v_{ij} \mid 1 \leq i \leq w, 1 \leq j \leq h]]) \} \\
ca & \in & Addr & = \{ \text{ cell addresses, e.g. B2, G\$6 etc. } \} \\
v, u & \in & Value & = Number + Error + ArrVal + FunVal \\
e & \in & Expr & = \{ \text{ formulas, e.g. } \texttt{=1+2} \} \\
fv & \in & FunVal & = \{ (sdf, [u_1, \ldots, u_k]) \} \\
\phi & & & \in Addr \to Expr \\
\sigma & & & \in Addr \to Value \\
\alpha & & & \in Expr \to Value \\
\rho & & & \in Addr \to Value \\
\gamma & & & \in Addr + Expr \to \mathbb{Z}^+
\end{array}
$$

Figure 8.2 – Extended semantic sets and environment maps used in the natural cost semantics [6].

The new semantic set $fv \in FunVal$ contains all function values i.e. closures denoted by the name of an SDF sdf and k arguments values where $0 \leq k \leq arity(sdf)$. We henceforth use $arity(sdf)$ to denote the arity of an SDF. The $Value$ set has been extended to include function values. A new environment ρ has been introduced which maps cell addresses to values. Coincidentally, it has the same signature as σ since ρ is used in the same way but looks up values in *function* sheets. Recall that function sheets are special sheets where SDFs can be defined using the `DEFINE` built-in function. However, ρ behaves differently in recursive SDFs where each recursive invocation gets its own fresh ρ map. After a function returns, one cannot refer to its local ρ anymore. Intuitively, we can view ρ as a function call stack frame in the implementation of ordinary programming languages. This analogy will be useful later when we introduce cost rules for evaluating SDFs. The α environment is unavailable in function sheets as array formulas cannot be defined in those types of sheet [23]. The final new environment γ maps a cell address ca or an array formula expression ae to a positive integer denoting the cost of evaluating the formula $\phi(ca)$ or the cost of evaluating an array formula expression ae.

8.1.2 Extending the Semantics

The cost semantics use unit costs to represent work and thus corresponds roughly to the number of rule applications. This simplified representation of costs suffices for our needs but we discuss alternative representations later to increase the precision of costs. To accommodate costs in the rules, we use a new evaluation judgement as given in equation (8.1). Given the σ and α environments, the expression e may evaluate to a value v at some cost c.

$$\sigma, \alpha \vdash e \Downarrow v, c \tag{8.1}$$

Next, we present the extended cost semantics in figure 8.3 for ordinary data sheets and explain each rule in turn. There are similar rules for function sheets where we replace σ with ρ and omit rules involving array formulas but we omit them here due to their similarity. Of course some rules, like application of an SDF, only use ρ as they can only be defined on function sheets.

$$\frac{}{\sigma, \alpha \vdash \texttt{n} \Downarrow n, 1}\ \text{(c1)} \qquad \frac{ca \notin dom(\sigma)}{\sigma, \alpha \vdash \texttt{ca} \Downarrow 0.0, 1}\ \text{(c2b)} \qquad \frac{ca \in dom(\sigma) \quad \sigma(ca) = v}{\sigma \vdash \texttt{ca} \Downarrow v, 1}\ \text{(c2v)}$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1, c_1 \quad v_1 \in Error}{\sigma, \alpha \vdash \texttt{IF}(e_1, e_2, e_3) \Downarrow v_1, 1 + c_1}\ \text{(c3e)} \qquad \frac{\sigma, \alpha \vdash e_1 \Downarrow 0.0, c_1 \quad \sigma, \alpha \vdash e_3 \Downarrow v, c_3}{\sigma, \alpha \vdash \texttt{IF}(e_1, e_2, e_3) \Downarrow v, 1 + c_1 + c_3}\ \text{(c3f)}$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \quad v_1 \neq 0.0 \quad \sigma, \alpha \vdash e_2 \Downarrow v, c_2}{\sigma, \alpha \vdash \texttt{IF}(e_1, e_2, e_3) \Downarrow v, 1 + c_1 + c_2}\ \text{(c3t)}$$

$$\frac{0.0 \leq v < 1.0}{\sigma, \alpha \vdash \texttt{RAND}() \Downarrow v, 1}\ \text{(c4)} \qquad \frac{\begin{array}{c} J \subseteq \{1, \ldots, n\} \\ \forall j \in J.\ \sigma, \alpha \vdash e_j \Downarrow v_j, c_j \quad v_i \in Error \end{array}}{\sigma, \alpha \vdash \texttt{F}(e_1, \ldots, e_n) \Downarrow v_i, 1 + \sum_{j \in J} c_j}\ \text{(c5e)}$$

$$\frac{\begin{array}{c} \sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \quad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n \\ \forall i.\ v_i \notin Error \end{array}}{\sigma, \alpha \vdash \texttt{F}(e_1, \ldots, e_n) \Downarrow f(v_1, \ldots, v_n), 1 + \sum_{j=1,n} c_j + work(f, v_1, \ldots, v_n)}\ \text{(c5v)}$$

$$\frac{\begin{array}{cc} (c_1, r_1) = ca_1 & (c_2, r_2) = ca_2 \\ (c_l, c_r) = sort(c_1, c_2) & (r_t, r_b) = sort(r_1, r_2) \\ w = c_r - c_l + 1 & h = r_b - r_t + 1 \end{array}}{\sigma, \alpha \vdash \texttt{ca}_1 : \texttt{ca}_2 \Downarrow Av(w, h, [[\sigma[c_l + i, r_t + j] \mid 1 \leq i \leq w, 1 \leq j \leq h]]), w \cdot h}\ \text{(c6)}$$

$$\frac{}{\sigma, \alpha \vdash ae[i,j] \Downarrow \alpha(ae)[i,j], 1} \text{ (c7)}$$

$$\frac{\begin{array}{c}\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \quad \ldots \quad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n \\ def(sdf) = (out, [in_1, \ldots, in_n], cells) \\ \rho'(in_1) = v_1 \quad \ldots \quad \rho'(in_n) = v_n \\ \forall ca \in dom(\rho') \setminus \{in_1, \ldots, in_n\}.\ \rho', \sigma \vdash \phi(ca) \Downarrow \rho'(ca), \gamma'(ca)\end{array}}{\sigma, \alpha \vdash sdf(e_1, \ldots, e_n) \Downarrow \rho'(out), 1 + \sum_{j=1,n} c_j + \sum_{ca \in dom(\gamma')} \gamma'(ca)} \text{ (c8)}$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow u_1, c_1 \quad \ldots \quad \sigma, \alpha \vdash e_k \Downarrow u_k, c_k}{\sigma, \alpha \vdash \texttt{CLOSURE}(sdf, e_1, \ldots, e_k) \Downarrow FunVal(sdf, [u_1, \ldots, u_k]), 1 + \sum_{j=1,k} c_j} \text{ (c9)}$$

$$\frac{\begin{array}{c}\sigma, \alpha \vdash e_0 \Downarrow FunVal(sdf, [u_1, \ldots, u_k]), c_0 \\ \sigma, \alpha \vdash e_n \Downarrow v_n, c_n \quad \ldots \quad \sigma, \alpha \vdash e_1 \Downarrow v_1, c_1\end{array}}{\sigma, \alpha \vdash \texttt{CLOSURE}(e_0, e_1, \ldots, e_n) \Downarrow FunVal(sdf, [u_1, \ldots, u_k, v_1, \ldots, v_n]), 1 + c_0 + \sum_{j=1,n} c_j} \text{ (c10)}$$

$$\frac{\begin{array}{c}\sigma, \alpha \vdash e_0 \Downarrow FunVal(sdf, [u_1, \ldots, u_k]), c_0 \\ \sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \quad \ldots \quad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n \\ def(sdf) = (out, [in_1, \ldots, in_{k+n}], cells) \\ \rho'(in_1) = u_1 \ldots \rho'(in_k) = u_k \quad \rho'(in_{k+1}) = v_1 \ldots \rho'(in_{k+n}) = v_n \\ \forall ca \in dom(\rho') \setminus \{in_1, \ldots, in_{k+n}\}.\ \rho', \sigma \vdash \phi(ca) \Downarrow \rho'(ca), \gamma'(ca)\end{array}}{\sigma, \alpha \vdash \texttt{APPLY}(e_0, e_1, \ldots, e_n) \Downarrow \rho'(out), 1 + c_0 + \sum_{j=1,n} c_j + \sum_{ca \in dom(\gamma')} \gamma'(ca)} \text{ (c11)}$$

Figure 8.3 – Cost semantic rules for Funcalc [6].

- Rules (c1) to (c7) are very similar to their non-cost counterparts except they now include a cost. Rules (c1) to (c2v), (c4) and (c7) all have constant cost.

- Rules (c3e), (c3f) and (c3t) handle the conditional `IF` function. In rule (c3e), the condition evaluates to an error value so the resulting cost is 1 plus the the cost c_1 of evaluating the conditional expression. Note that the cost does not include the cost of the branches since neither is ever evaluated. Rules (c3f) and (c3t) handle the true and false cases and including the cost of evaluating the respective branches. Generally, our rules include an additional unit cost so successive rule applications are monotonically increasing.

- Rules (c5e) and (c5v) handle calls to intrinsic functions. The rules are quite different from the simple semantics and require explanation. To evaluate a call to an intrinsic function where an argument may evaluate to an error value, we evaluate a subset of arguments

$\{e_j \mid j \in J\}$ that evaluate to values at some costs v_j, c_j. If there is some $i \in J$ for which v_i is an error value, the result of calling the function evaluates to v_i consistent with treatment of error value as first-class. The total cost is one plus the cost of evaluating the subset of arguments.

The additional complexity of this rule accommodates several implementations of function evaluation. By using J, we leave it up to the implementation to decide how to evaluate the arguments. For example, left-to-right evaluation could simply evaluate expressions $e_1, e_2, \ldots$ and so forth until one evaluates to an error or all argument expressions have been evaluated, and similarly for right-to-left evaluation. One may also choose $J = \{1, \ldots, n\}$ to evaluate all arguments in an arbitrary order before checking for an error value. This also admits both strict and non-strict evaluation as well as parallel evaluation of the expressions. How J is chosen ultimately affects the total cost of calling an intrinsic function.

- Rule (c6) is unchanged from its non-cost counterpart but now evaluates to an array value at cost $w \cdot h$ corresponding to the size of the array. The cost is overly pessimistic. Any sensible implementation would return a view of the cell area instead of allocating a new array as Funcalc currently does.

- Rule (c8) handles the evaluation of an SDF and uses ρ to look up values from cell addresses in function sheets. Let us consider the premises from top to bottom. The first premise states the argument expressions may evaluate to some values at some costs. The second premise says the definition of the sdf is the addresses of its output cell out, its n input cells and a set of intermediate cells $cells$. The third premise states that the values of the arguments to the SDF can be found in the input cells via ρ'. Here, ρ' corresponds to the one *local* to this particular, possibly recursive, call to the SDF. The last premise states that for all cell addresses in the domain of ρ', excluding the addresses of the input cells, we can evaluate their formula expressions to some value at some cost. Since ρ' is local to the function application, its domain entails only cells relevant to the function's application. Given these premises, the conclusion may evaluate to the value in the output cell out of the sdf at cost one plus the cost of evaluating its argument expressions

plus the cost of evaluating each cell address in the domain of γ' which corresponds to $dom(\rho') \setminus \{in_1, \ldots, in_n\}$ and is also local to the function application.

We have strived for a semantics that gives an implementation ample freedom. This is especially important for rule (c8) as the actual implementation of SDFs compiles to CIL bytecode behind the scenes. In Funcalc, we do not actually read the result of calling the SDF in its output cell, although it is very intuitive, but retrieve the result from the execution of the bytecode instructions of the SDF. However, the end result is semantically equivalent to reading the value in the output cell. If SDFs were instead interpreted, we would retrieve the result from output cell.

- Rule (c9) handles closure creation and states the conclusion may evaluate to a function value of the given sdf and a list of values $u_1, \ldots, u_k$ of the argument expressions to the call to `CLOSURE`. Its cost is one plus the cost of evaluating the argument expressions.

- Rule (c10) handles partial closure application. It states e_0 may evaluate to a function value with bound values $u_1, \ldots, u_k$ and the argument expressions $e_1, \ldots, e_n$ may evaluate to values at some costs, then the conclusion may evaluate to a new function value which closes over the early-bound values $u_1, \ldots, u_k$ and the new values of the premise. Its cost is one plus the cost of evaluating the function value and the remaining argument expressions.

- Finally, we have rule (c11) for closure application via the `APPLY` built-in. The rule is similar to rule (c8) except e_0 is expected to evaluate to a function value at some cost c_0 with k early-bound arguments for $0 \leq k \leq arity(sdf)$. The remaining n argument expressions are given to `APPLY` such that the application of the SDF, where $arity(sdf) = k + n$, can succeed.

Let us consider the domain of ρ in slightly more detail in relation to the total cost of calling an SDF. It is only necessary for $dom(\rho)$ to contain the cells that are needed to compute the value of the output cell $\rho(out)$. By excluding cells not needed by the output cell from $dom(\rho)$ and hence $dom(\gamma)$, we avoid cells contributing to the total cost of the SDF if they are not used.

	A	B
1	=DEFINE("func", B5, B2, B3)	
2	"n="	
3	"p="	
4		=B2*B2*B3*B3
5	"result="	=IF(RAND()<B2, B3, B4)

Figure 8.4 – The cost of an SDF may change depending on control flow. It may not be necessary to evaluate the intermediate cell B4 depending on the outcome of the condition of the result cell B5.

Consider the SDF definition of figure 8.4. The total cost depends on the value of input cell B2 and the call to `RAND` in B5 since B4 may not be needed to compute the output cell in some cases.

8.1.3 First-Order Intrinsic Functions

In this section, we present the big-step evaluation rules for a meaningful subset of first-order functions in Funcalc as omit rules for some of the intrinsic functions. For example, the `EXTERN` function returns the result of a call to an external DLL. While the returned value is given by a plain C# `object` type, its cost is hard to define in general. The call may perform any operation from querying a database to initiating some long-running, unknown computation that we have insufficient knowledge to approximate. Alternatively, we could give meaningful rules for some common uses for `EXTERN` such as the methods in the .NET libraries or let users explicitly annotate costs, but we forgo this here. Rules trivially similar to each other such as `ASIN`, `ACOS` and `ATAN` are not all shown and we just give a single rule to represent them all. Like for the extended semantic rules of section 8.1.1, we focus only on ordinary, interpreted sheets, as the rules for function sheets are analogous. To keep the rules using array values compact, we introduce the following short-hand notation.

$$Av(w,h,[\![v_{ij}]\!]) \triangleq ArrVal(w,h,[\![v_{ij} \mid 1 \leq i \leq w, 1 \leq j \leq h]\!])$$

Additionally, when a conclusion needs to refer to an array value given in a premise, we use the notation $arr = [\![v_{ij}]\!]$ to assign the values of an array value to arr in the premise and refer to arr in the conclusion. It should be clear from context what the assigned array value refers to. We omit "error rules" for cases where a function argument evaluates to

an error value and refer the reader to rule (c5e) in figure 8.3. The cost rules for the first-order functions are given in figure 8.5.

$$\frac{v \in Number}{\sigma, \alpha \vdash \texttt{NOW}() \Downarrow v, 1}\ \text{(now)} \qquad \frac{}{\sigma, \alpha \vdash \texttt{PI}() \Downarrow \pi, 1}\ \text{(pi)} \qquad \frac{}{\sigma, \alpha \vdash \texttt{NA}() \Downarrow \texttt{\#NA}, 1}\ \text{(na)}$$

$$\frac{\sigma, \alpha \vdash e \Downarrow v, c \quad v \in Number}{\sigma, \alpha \vdash \texttt{ABS}(e) \Downarrow |v|, 1 + c}\ \text{(abs)} \qquad \frac{\sigma, \alpha \vdash e \Downarrow v, c \quad v \in Number}{\sigma, \alpha \vdash \texttt{ASIN}(e) \Downarrow asin(v), 1 + c}\ \text{(asin)}$$

$$\frac{\sigma, \alpha \vdash e \Downarrow v, c \quad v = 0}{\sigma, \alpha \vdash \texttt{NOT}(e) \Downarrow 1, 1 + c}\ \text{(not-1)} \qquad \frac{\sigma, \alpha \vdash e \Downarrow v, c \quad v \neq 0}{\sigma, \alpha \vdash \texttt{NOT}(e) \Downarrow 0, 1 + c}\ \text{(not-2)}$$

$$\frac{\begin{array}{cc}\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 & \sigma, \alpha \vdash e_2 \Downarrow v_2, c_2 \\ v_1 \in Number & v_2 \in Number\end{array}}{\sigma, \alpha \vdash \texttt{CEILING}(e_1, e_2) \Downarrow ceiling(v_1, v_2), 1 + c_1 + c_2}\ \text{(ceiling)}$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \quad \sigma, \alpha \vdash e_2 \Downarrow v_2, c_2}{\sigma, \alpha \vdash e_1 = e_2 \Downarrow v_1 = v_2, 1 + c_1 + c_2}\ \text{(equal)}$$

$$\frac{\begin{array}{c}J \subseteq \{1, \ldots, n\} \\ \forall j \in J.\, v_j \in Number \qquad \forall j \in J.\, \sigma, \alpha \vdash e_j \Downarrow v_j, c_j \qquad \exists i \in J.\, v_i = 0\end{array}}{\sigma, \alpha \vdash \texttt{AND}(e_1, \ldots, e_n) \Downarrow 0, 1 + \sum_{j \in J} c_j}\ \text{(and-false)}$$

$$\frac{\begin{array}{c}J = \{1, \ldots, n\} \\ \forall j \in J.\, v_j \in Number \wedge v_j \neq 0 \quad \forall j \in J.\, \sigma, \alpha \vdash e_j \Downarrow v_j, c_j\end{array}}{\sigma, \alpha \vdash \texttt{AND}(e_1, \ldots, e_n) \Downarrow 1, 1 + \sum_{j \in J} c_j}\ \text{(and-true)}$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \quad \ldots \quad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n}{\sigma, \alpha \vdash \texttt{SUM}(e_1, \ldots, e_n) \Downarrow \sum_{i=1}^{n} v_i, 1 + \sum_{i=1}^{n} c_i}\ \text{(sum)}$$

$$\frac{\begin{array}{ccc}\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 & \sigma, \alpha \vdash e_2 \Downarrow v_2, c_2 & \sigma, \alpha \vdash e_3 \Downarrow v_3, c_3 \\ & v_2 \in Number \wedge v_2 \geq 0 & v_3 \in Number \wedge v_3 \geq 0\end{array}}{\sigma, \alpha \vdash \texttt{CONSTARRAY}(e_1, e_2, e_3) \Downarrow Av(\lfloor v_3 \rfloor, \lfloor v_2 \rfloor, [[v_1 \mid i \leq v_2, j \leq v_3]]), 1 + c_1 + c_2 + c_3 + v_3 \cdot v_2}\ \text{(const-array)}$$

$$\frac{\sigma, \alpha \vdash e_0 \Downarrow s, c_0 \quad s \in Number \wedge 1 \leq s \leq n \quad \sigma, \alpha \vdash e_{\lfloor s \rfloor} \Downarrow v_s, c_s}{\sigma, \alpha \vdash \texttt{CHOOSE}(e_0, e_1, \ldots, e_n) \Downarrow v_s, 1 + c_0 + c_s}\ \text{(choose)}$$

$$\frac{\sigma, \alpha \vdash e \Downarrow Av(w, h, \left[\left[v_{ij}\right]\right]), c}{\sigma, \alpha \vdash \texttt{COLUMNS}(e) \Downarrow w, 1 + c}\ \text{(columns)}$$

$$\frac{\begin{array}{c}\sigma,\alpha \vdash e_1 \Downarrow Av(w,h,\left[\left[v_{ij}\right]\right]),c_1 \\ \sigma,\alpha \vdash e_2 \Downarrow v_2,c_2 \quad \sigma,\alpha \vdash e_3 \Downarrow v_3,c_3 \\ v_2 \in Number \wedge 1 \leq v_2 < w+1 \quad v_3 \in Number \wedge 1 \leq v_3 < h+1\end{array}}{\sigma,\alpha \vdash \texttt{INDEX}(e_1,e_2,e_3) \Downarrow v_{\lfloor v_3 \rfloor \lfloor v_2 \rfloor},1+c_1+c_2+c_3} \text{ (index)}$$

$$\frac{\begin{array}{cc}\sigma,\alpha \vdash e_1 \Downarrow Av(w,h,\left[\left[v_{ij}\right]\right]),c_1 & arr = \left[\left[v_{ij}\right]\right] \\ \sigma,\alpha \vdash e_2 \Downarrow v_2,c_2 \quad \sigma,\alpha \vdash e_4 \Downarrow v_4,c_4 & \sigma,\alpha \vdash e_3 \Downarrow v_3,c_3 \quad \sigma,\alpha \vdash e_5 \Downarrow v_5,c_5 \\ v_2 \in Number \quad v_4 \in Number & v_3 \in Number \quad v_5 \in Number \\ 1 \leq v_4 < h+1 \quad 1 \leq v_2 < h+1 & 1 \leq v_3 < w+1 \quad 1 \leq v_5 < w+1 \\ h' = \lfloor v_4 \rfloor - \lfloor v_2 \rfloor + 1 & w' = \lfloor v_5 \rfloor - \lfloor v_3 \rfloor + 1 \\ \multicolumn{2}{c}{r = Av(w',h',[[arr[i,j] \mid v_3 \leq i \leq v_5, v_2 \leq j \leq v_4]])}\end{array}}{\sigma,\alpha \vdash \texttt{SLICE}(e_1,e_2,e_3,e_4,e_5) \Downarrow r,1+c_1+c_2+c_3+c_4+c_5+w' \cdot h'} \text{ (slice)}$$

$$\frac{\sigma,\alpha \vdash e \Downarrow v,c \quad v \in Error}{\sigma,\alpha \vdash \texttt{ISERROR}(e) \Downarrow 1,1+c} \text{ (iserror-true)} \quad \frac{\sigma,\alpha \vdash e \Downarrow v,c \quad v \notin Error}{\sigma,\alpha \vdash \texttt{ISERROR}(e) \Downarrow 0,1+c} \text{ (iserror-false)}$$

$$\frac{\sigma,\alpha \vdash e \Downarrow v,c \quad v \in ArrVal}{\sigma,\alpha \vdash \texttt{ISARRAY}(e) \Downarrow 1,1+c} \text{ (isarray-true)} \quad \frac{\sigma,\alpha \vdash e \Downarrow v,c \quad v \notin ArrVal}{\sigma,\alpha \vdash \texttt{ISARRAY}(e) \Downarrow 0,1+c} \text{ (isarray-false)}$$

$$\frac{\sigma,\alpha \vdash e_1 \Downarrow v_1,c_1 \quad \ldots \quad \sigma,\alpha \vdash e_n \Downarrow v_n,c_n}{\sigma,\alpha \vdash \texttt{MAX}(e_1,\ldots,e_n) \Downarrow max(v_1,\ldots,v_n),1+\sum_{j=1}^{n} c_j} \text{ (max)}$$

$$\frac{\sigma,\alpha \vdash e \Downarrow Av(w,h,\left[\left[v_{ij}\right]\right]),c \quad arr = \left[\left[v_{ij}\right]\right]}{\sigma,\alpha \vdash \texttt{TRANSPOSE}(e) \Downarrow Av(h,w,[[arr[j,i]]]),1+c+w \cdot h} \text{ (transpose)}$$

$$\frac{\sigma,\alpha \vdash e_1 \Downarrow v_1,c_1 \quad \ldots \quad \sigma,\alpha \vdash e_n \Downarrow v_n,c_n}{\sigma,\alpha \vdash \texttt{AVERAGE}(e_1,\ldots,e_n) \Downarrow \frac{1}{n}\sum_{i=1}^{n} v_i,1+\sum_{i=1}^{n} c_i} \text{ (average)}$$

$$\frac{\sigma,\alpha \vdash e_1 \Downarrow v_1,c_1 \quad \ldots \quad \sigma,\alpha \vdash e_n \Downarrow v_n,c_n}{\sigma,\alpha \vdash \texttt{HARRAY}(e_1,\ldots,e_n) \Downarrow Av(n,1,[[v_1,\ldots,v_n]]),1+\sum_{i=1}^{n} c_i+n} \text{ (harray)}$$

$$\frac{\begin{array}{c}\forall i,j.\, height(v_i) = height(v_j) \quad w = \sum_{i=1}^{n} width(v_i) \\ \sigma,\alpha \vdash e_1 \Downarrow v_1,c_1 \quad \ldots \quad \sigma,\alpha \vdash e_n \Downarrow v_n,c_n\end{array}}{\sigma,\alpha \vdash \texttt{HCAT}(e_1,\ldots,e_n) \Downarrow Av(w,height(v_1),[[v_1 : v_2 : \ldots : v_n]]),1+\sum_{i=1}^{n} c_i+n} \text{ (hcat)}$$

Figure 8.5 – Cost semantics for a subset of Funcalc's built-in first-order functions [6].

- Rule (now) returns the number of fractional days since the 30^{th} of December, 1899. It states that a call to `NOW` may evaluate to a value v at cost 1 given that v is a proper number.

- Rule (pi) returns the mathematical constant π. It states it may evaluate to a value v at cost 1 given that $v = \pi$.

- Rule (na) states a call to `NA` may evaluate to the error value `#NA` at cost 1. This function is primarily used to denote late-bound arguments in closures.

- Rule (abs) states if e may evaluate to the number v at cost c then the call evaluates to the absolute value of v.

- Rule (asin) is similar to rule (abs) but may instead evaluate to the result of a call to the actual inverse trigonometric function *asin*.

- The `NOT` function handles boolean negation. The rules (not-1) and (not-2) handle the two different outcomes of the function. Rule (not-1) states that if e may evaluate to zero at some cost then the call may evaluate to 1. Rule (not-2) handles the case where the value is different from zero in which case the result may evaluate to 0. The cost in both cases is one plus the cost of evaluating the argument expression.

- The `CEILING` function returns the least integer greater than or equal to its argument, i.e. $\lceil x \rceil$. Its rule (ceiling) states that if its two argument expressions may evaluate to numbers then the conclusion may evaluate to a call to actual ceiling function with the two numbers as arguments. The function rounds v_1 toward $+\infty$ when $v_2 > 0$ and toward $-\infty$ when $v_2 < 0$. It returns an error value if $v_2 = 0$ but we omit its rule.

- Rule (equal) may evaluate to the equality comparison between two number values. The actual implementation of equality is slightly more involved as it also takes `null` values and object equality of different classes into account.

- Rules (and-false) and (and-true) are short-circuiting functions that return 1 if all expressions are true (non-zero) and 0 if just one expression is false (zero). They borrow notation from rules (c5e) and (c5v) from section 8.1.1 to give an implementation freedom to choose how to evaluate the arguments. In rule (and-false), one may pick some subset of indices J where all the corresponding expressions may evaluate to numbers. If there exists an index i for which the expression may evaluate to zero, the result of calling `AND` is zero. The cost is one plus the cost of evaluating the expressions given by the set of indices in J. Rule (and-true) handles the case where the values of all argument expressions may evaluate to a non-zero number value. The rule for `OR` is analogous. The total work is proportional to the subset of expressions evaluated plus one.

- Rule (sum) says that if all argument expressions may evaluate to number values then the function call may evaluate to the sum of those values. The cost is one plus the cost of evaluating each argument expression. In Funcalc, functions like `SUM` and `AVERAGE` can accept a combination of numbers and array values and the result of the different calls to `SUM` are all 21 in the examples below. We choose not to complicate the rules further by disregarding array values in the arguments but one could imagine some sort of flattening function applied to each value in the summation in the conclusion of rule (sum) to account for array values.

```
=SUM(1, 2, 3, 4, 5, 6)
=SUM(HCAT(1, 2, 3, 4, 5, 6))
=SUM(VCAT(1, 2, 3, 4, 5, 6))
=SUM(HCAT(1, 2), 3, VCAT(4, 5, 6))
```

- Function `CONSTARRAY` creates an array value containing a single value as its elements. Rule (const-array) says that if expression e_1 may evaluate to a value at some cost c_1 and expressions e_2 and e_3 may evaluate to non-negative numbers, then the call may evaluate to an array value of size $v_3 \cdot v_2$ with v_1 as the value of each element. The cost reflects that it is only necessary to evaluate e_1 once.

- The `CHOOSE` function allows users to choose a specific value using the first argument as a selector, given it evaluates to a one-based index which picks the corresponding value from the remaining arguments. For example, `CHOOSE(2, 1, 2, 3)` would return 2.

  Rule (choose) states that if e_0 may evaluate to a positive number $s \in [1, n[$ and the expression $e_{\lfloor s \rfloor}$ may evaluate to a value v_s at cost c_s, then the call may evaluate to v_s at cost $1 + c_0 + c_s$. Note s may lie in a wider interval than required since it is truncated. Like `IF`, `CHOOSE` is non-strict so it only evaluates the selected arguments or none in case the selector is an error value. A more general rule would adopt the same approach we used for the rules for `AND` and have $J = \{0, s\}$ as a special case.

- The `COLUMNS` function returns the width of its argument. There is a corresponding function called `ROWS` for height. Rule (columns) states that if e may evaluate to an array value then a call to `COLUMNS` may evaluate to the width of that array value. The cost is overly pessimistic since for some simple cases we could inspect the width of the argument without having to evaluate it.

- We can use the `INDEX` function to index a cell area or array value. Rule (index) states that if e_1 may evaluate to an array value and e_2 and e_3 may evaluate to numbers within the bounds of the array value, then the conclusion may evaluate to the value at index $(\lfloor v_3 \rfloor, \lfloor v_2 \rfloor)$. Like rule (columns) the cost is overly pessimistic and like rule (choose) the indices are truncated towards zero.

- The `SLICE` function returns a slice or sub-array of an array value or cell area argument. In rule (slice), the premises state that e_1 may evaluate to an array value, expressions e_2 and e_4 may evaluate to start- and end column indices and expressions e_3 and e_5 may evaluate to start- and end row indices. The indices collectively delimit a sub-array within the input array. The conclusion may then evaluate to a new array value slice of the original array. The sub-array's dimensions w' and h' are computed from the row and column indices. The cost is one plus evaluating the four indices plus the cost of evaluating the input array value plus the size of the new array.

The total cost is also slightly pessimistic. In Funcalc, `SLICE` produces a *view* without actually allocating a new array. An implementation may however choose to allocate a new array.

- Rule (iserror-true) states that if e may evaluate to an error value v then the call may evaluate to 1 (true). Rule (iserror-false) is complementary and handles the case where v is not an error value.

- Rule (isarray-true) and (isarray-false) are analogous to rules (iserror-true) and (iserror-false) but check whether the argument is an array value.

- Rule (max) states that if all the argument expressions may evaluate to numbers at some costs, then a call to `MAX` may evaluate to the maximal value among those values. The rule for `MIN` is analogous.

- Rule (transpose) states that if the argument expression may evaluate to an array value of size w columns and h rows with cost c, then the call may evaluate to a transposed array value of size h columns and w rows. Notice elements are accessed in the result array with swapped indices swapped. The work is one plus the cost c and the size of the resultant array.

- Rule (average) is similar to rule (sum) but the conclusion may instead evaluate to the average of the input values.

- Rule (harray) states that if the arguments may evaluate to a set of values at associated costs then the call may evaluate to a single-row array of those values. This is consistent with the behaviour of `HARRAY` which simply puts the values of the evaluated expressions inside an array. The expression `=HARRAY(1, HARRAY(2, 3))` will yield an array value of width 2 and height 1 where the first element is the value 1 and the second element is an array with values 2 and 3. The rule for `VARRAY` is similar. The n in the cost of the conclusion denotes the cost of allocating the new array of size n.

- Rule (hcat) is closely related to the rule for `HARRAY` but concatenates its arguments. Its premises state the expressions may evaluate to values at some associated costs as in rule (harray). The width w of the new array value is the sum of the widths of all its

arguments. We use the auxiliary helper functions *width* and *height* to retrieve the height and widths of the arguments, defined as follows. Only the *width* function is shown as the definition of *height* is analogous.

$$width(v) = \begin{cases} w & \text{if } v = Av(w, h, [[v_{ij}]]) \\ 1 & \text{otherwise} \end{cases}$$

We also require all argument values have the same height otherwise we would not be able to properly concatenate them horizontally. Given these premises, the conclusion may evaluate to an array value of width w and the height of the arguments. The elements of the array value are the concatenation of the evaluated expressions using the semantic colon concatenation operator. The n in the cost of the conclusion denotes the cost of concatenation which is assumed to be constant time for an efficient implementation of array concatenation.

To understand the difference between `HCAT` and `HARRAY`, calling the same expression as before with `HCAT`, i.e. `=HCAT(1, HARRAY(2, 3))`, yields an array value of width 3 and height 1. This would also be the case if we replaced the inner call to `HARRAY` with a call to `HCAT`. The rule for `VCAT` is similar and has been omitted for brevity.

8.1.4 Higher-Order Intrinsic Functions

The rules for the higher-order functions of Funcalc build on rule (c5v) for application of built-in functions from our extended semantics. One significant difference is that most higher-order functions apply the supplied function value multiple times so we introduce quantification over the fresh environment ρ' for each application. Recall that ρ' can be thought of as a stack frame for the current function application. For example, the function `TABULATE` creates an array where each element is the result of calling a function with the indices of each element and whose size is specified by the user. We quantify over the fresh environment with the current position (i, j) as ρ'_{ij}. Similarly, we also quantify over the cost environment γ' as γ'_{ij}. Evaluating the following expression

```
=TABULATE(CLOSURE("+"), 2, 2)
```

would give us an 2 by 2 array where each element is the addition of its one-based position in the array. As an illustration of rules for higher-order functions, let us define the rule for TABULATE and helpful auxiliary functions, before presenting the remaining rules.

$$\frac{\begin{array}{c}\sigma,\alpha \vdash e_1 \Downarrow FunVal(sdf,[u_1,\ldots,u_k]),c_1 \\ def(sdf) = (out,[in_1,\ldots,in_{k+2}],cells) \\ \sigma,\alpha \vdash e_2 \Downarrow v_2,c_2 \quad v_2 \in Number \wedge h = \lfloor v_2 \rfloor \geq 0 \\ \sigma,\alpha \vdash e_3 \Downarrow v_3,c_3 \quad v_3 \in Number \wedge w = \lfloor v_3 \rfloor \geq 0 \\ \forall i,j.\, \rho'_{ij}(in_1) = u_1 \;\; \ldots \;\; \rho'_{ij}(in_k) = u_k \quad \rho'_{ij}(in_{k+1}) = i \quad \rho'_{ij}(in_{k+2}) = j \\ \forall i,j.\, \forall ca \in dom(\rho'_{ij}) \setminus \{in_1,\ldots,in_{k+2}\}.\, \sigma,\alpha \vdash \phi(ca) \Downarrow \rho'_{ij}(ca),\gamma'_{ij}(ca) \\ \forall i,j.\, v_{ij} = \rho'_{ij}(out) \quad c_4 = \sum_{i,j}^{dom(\gamma'_{ij})} \sum_{ca} \gamma'_{ij}(ca)\end{array}}{\sigma,\alpha \vdash \mathsf{TABULATE}(e_1,e_2,e_3) \Downarrow Av(w,h,\left[\left[v_{ij}\right]\right]),1+c_1+c_2+c_3+c_4} \text{ (tabulate)}$$

Considering the premises from top to bottom, they state that e_1 may evaluate to a function value at cost c_1 expecting two more arguments as indicated by the definition of the sdf where its last argument is in_{k+2}. Expressions e_2 and e_3 may evaluate to non-negative numbers h and w for the rows and columns, truncated towards zero. We then postulate $w \cdot h$ environments ρ'_{ij} where $1 \leq i \leq w$ and $1 \leq j \leq h$, one for each application of the function value.

The first quantified premise states the input cells should contain the early-bound values $[u_1,\ldots,u_k]$ of the function value except the last two arguments must be the indices i and j indices of the current position. The next quantified premise is similar to that of rules (c8) and (c11) and states for all cell addresses ca in the domain of the quantified environment ρ'_{ij}, excluding the set of input cells, the expression of that cell address may evaluate to the value given by environment ρ'_{ij} at some cost given by γ'_{ij}. The final premise states each function application evaluates to the value v_{ij} of the the output cell. As before the quantification of ρ and γ over i and j can be thought of as a fresh stack frame for each function call. The call to TABULATE then evaluates to an array value of size $w \cdot h$ whose elements are v_{ij}. The cost is 1 plus the cost for evaluating the function value, the array size expressions at costs c_2 and c_3, and the sum of the costs of applying the sdf for every index (i,j) for each cell address in the domain of the quantified environment as given by c_4.

The higher-order functions in Funcalc all accept function values and will also need to store values in the input cells, evaluate each cell address in $dom(\rho')$, read the result of the function in its output cell, and compute the total cost of function application. In the interest of reducing repetition, we define an auxiliary function $apply$ as follows.

$$apply_{\sigma,\alpha}(sdf, [u_1, \ldots, u_k], a_0, \ldots, a_n, r, c) \triangleq \begin{cases} \rho'(in_1) = u_1 \ \ldots \ \rho'(in_k) = u_k \quad \rho'(in_{k+1}) = a_0 \ \ldots \ \rho'(in_{k+n}) = a_n \\ \forall ca \in dom(\rho') \setminus \{in_1, \ldots, in_{k+n}\}. \rho', \sigma \vdash \phi(ca) \Downarrow \rho'(ca), \gamma'(ca) \\ r = \rho'(out) \quad c = \sum_{ca}^{dom(\gamma')} \gamma'(ca) \end{cases}$$

The $apply$ function takes as arguments the sdf to apply, a list of early-bound argument values $u_1, \ldots, u_k$, a variable number of late-bound arguments $a_0, \ldots, a_n$ and a variable for the result r and cost c as a result of the function application. Note the resulting value r and cost c are passed back out of the $apply$ function.

Recursive, Higher-order Functions

This definition of $apply$ makes sense for functions like `TABULATE` where we can refer to specific function applications using $apply_{\sigma,\alpha}$ and their results using r_{ij} and c_{ij}. However, the situation is more complex for recursive functions like `REDUCE` where intermediate function applications operate on values, not expressions. The function takes a function value, a starting value and an array value and applies the function value to reduce the array values to a scalar as in functional programming. For example, `=REDUCE(CLOSURE("+"), 0, HCAT(1, 2, 3))` would sum the values in the array.

Recursive functions would be less problematic if we ignored costs and simply operated on expressions throughout the recursive calls. Since we *do* consider costs, evaluating the same expressions for successive recursive calls would grossly overestimate the total cost as any sensible implementation would only evaluate the argument expressions at the top-level call once. To account for this, we introduce a new judgement $\sigma, \alpha \vdash_v v \Downarrow w, c$ that operates on values (note the subscript v on the turnstile) instead of expressions in order to handle the intermediate value-

based computations of recursive functions. The judgement states that given the usual environments σ and α, some intermediate value v may evaluate to a another value w at cost c. This allows us to evaluate the top-level argument expressions of the call once and use their values in subsequent recursive calls.

Luckily, the REDUCE function is the only function in Funcalc that needs a recursive rule definition. We need four rules in total: a top-level rule for the initial application that operates on expressions; an inductive rule that operates on values; and two base cases for handling an odd number of arguments and the empty array value. Let us start with the top-level rule.

$$\frac{\begin{array}{cc} \sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 & \\ v_1 = \mathit{FunVal}(\mathit{sdf}, [u_1, \ldots, u_k]) & \sigma, \alpha \vdash e_3 \Downarrow v_3, c_3 \\ k = \mathit{arity}(\mathit{sdf}) - 2 & v_3 \in \mathit{ArrVal} \\ \sigma, \alpha \vdash e_2 \Downarrow v_2, c_2 & \sigma, \alpha \vdash_v \mathsf{REDUCE}(v_1, v_2, v_3) \Downarrow r, c_r \end{array}}{\sigma, \alpha \vdash \mathsf{REDUCE}(e_1, e_2, e_3) \Downarrow r, 1 + c_1 + c_2 + c_3 + c_r} \text{ (reduce)}$$

The expression-based rule (reduce) states expression e_1 may evaluate to a function value and e_2 to an initial value for the reduction. Expression e_3 may evaluate to an array v_3 which is passed as an argument to the value-based reduction rule. The conclusion may then evaluate to the result r of the value-based reduction.

The inductive, value-based rule (reduce-inductive) states that v_3 decomposes into two arrays v_l and v_r at some cost c_d. This can be an arbitrary decomposition as chosen by an implementation. Given an identity element and an associative, binary function, a reduction may e.g. proceed from left to right or decomposed as a tree. Notice we pass the result r_l of the reduction of the left decomposed array v_l to the reduction of the right decomposed array v_r. The reason is purely semantic and will become apparent shortly.

$$\frac{\begin{array}{cc} \sigma, \alpha \vdash_v v_3 = v_l : v_r, c_d & v_l \in \mathit{ArrVal} \wedge v_r \in \mathit{ArrVal} \\ v_1 \in \mathit{FunVal} & \sigma, \alpha \vdash_v \mathsf{REDUCE}(v_1, v_2, v_l) \Downarrow r_l, c_l \\ v_2 \in \mathit{Number} & \sigma, \alpha \vdash_v \mathsf{REDUCE}(v_1, r_l, v_r) \Downarrow r, c_r \end{array}}{\sigma, \alpha \vdash_v \mathsf{REDUCE}(v_1, v_2, v_3) \Downarrow r, 1 + c_3 + c_l + c_r + c_d} \text{ (reduce-inductive)}$$

The first base case rule (reduce-base-odd) accounts for an odd number of arguments which results in a call with a single-element array that we cannot decompose. Given the single-element array value v_3, we apply the sdf from v_1 to the starting value v_2 and the single element v_{11}

of v_3. In rule (reduce-inductive), if we did not thread the result through the left and right decomposition of the array argument rule (reduce-base-odd) might be applied more than once which in turn would cause the starting value v_2 to also be used more than once, yielding an incorrect result.

$$\frac{v_1 \in FunVal \qquad v_3 = Av(1, 1, [[v_{11}]]) \qquad apply_{\sigma,\alpha}(sdf, [u_1, \ldots, u_k], v_2, v_{11}, r, c)}{\sigma, \alpha \vdash_v \texttt{REDUCE}(v_1, v_2, v_3) \Downarrow r, 1 + c} \text{ (reduce-base-odd)}$$

Rule (reduce-base-empty) simply returns the starting value v_2 of the reduction if passed the empty array which is used for an even number of arguments to the call to REDUCE.

$$\frac{v_1 \in FunVal \qquad v_2 \in Number \qquad v_3 = Av(0, 0, [[]])}{\sigma, \alpha \vdash_v \texttt{REDUCE}(v_1, v_2, v_3) \Downarrow v_2, 1} \text{ (reduce-base-empty)}$$

Remaining Higher-Order Functions

For the remaining higher-order functions, we introduce some special notation for referring to single rows or columns within an array value $Av(w, h, \left[\left[v_{ij}\right]\right])$. We use $[[v_{i*}]]$ to refer to the i^{th} column and $\left[\left[v_{*j}\right]\right]$ to refer to the j^{th} row.

$$\frac{\begin{array}{c}\sigma, \alpha \vdash e_1 \Downarrow Av(w_1, h_1, \left[\left[v^1_{ij}\right]\right]), c_1 \qquad \ldots \qquad \sigma, \alpha \vdash e_n \Downarrow Av(w_n, h_n, \left[\left[v^n_{ij}\right]\right]), c_n \\ \sigma, \alpha \vdash e_0 \Downarrow FunVal(sdf, [u_1, \ldots, u_k]), c_0 \qquad k = arity(sdf) - n \\ \forall k, m.\, w_k = w_m \wedge h_k = h_m \qquad \forall i, j.\, apply_{\sigma,\alpha}(sdf, [u_1, \ldots, u_k], v^1_{ij}, \ldots, v^n_{ij}, r_{ij}, t_{ij})\end{array}}{\sigma, \alpha \vdash \texttt{MAP}(e_0, e_1, \ldots, e_n) \Downarrow Av(w_1, h_1, \left[\left[r_{ij}\right]\right]), 1 + c_0 + \sum_{k=1}^{n} c_k + \sum_{ij} t_{ij}} \text{ (map)}$$

$$\frac{\begin{array}{cc}\sigma, \alpha \vdash e_1 \Downarrow FunVal(sdf, [u_1, \ldots, u_k]), c_1 & \sigma, \alpha \vdash e_2 \Downarrow Av(w, h, \left[\left[v_{ij}\right]\right]), c_2 \\ k = arity(sdf) - h & \forall i.\, apply_{\sigma,\alpha}(sdf, [u_1, \ldots, u_k], [[v_{i*}]], r_{i1}, c_i)\end{array}}{\sigma, \alpha \vdash \texttt{COLMAP}(e_1, e_2) \Downarrow Av(w, 1, [[r_{i1}]]), 1 + c_1 + c_2 + \sum_i c_i} \text{ (colmap)}$$

$$\frac{\begin{array}{cc}\sigma, \alpha \vdash e_0 \Downarrow FunVal(sdf, [u_1, \ldots, u_k]), c_0 & \sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \ \ldots \ \sigma, \alpha \vdash e_n \Downarrow v_n, c_n \\ k = arity(sdf) - 1 & \forall i.\, apply_{\sigma,\alpha}(sdf, [u_1, \ldots, u_k], v_i, r_i, t_i)\end{array}}{\sigma, \alpha \vdash \texttt{COUNTIF}(e_0, e_1, \ldots, e_n) \Downarrow \sum\{1 \mid r_i = 1 \wedge i = \{1, \ldots, n\}\}, 1 + c_0 + \sum_{j=1}^{n} c_j + \sum_{i=1}^{n} t_i} \text{ (countif)}$$

$$\frac{\begin{array}{cc}\sigma, \alpha \vdash e_0 \Downarrow FunVal(sdf, [u_1, \ldots, u_k]), c_0 & \sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \ \ldots \ \sigma, \alpha \vdash e_n \Downarrow v_n, c_n \\ k = arity(sdf) - 1 & \forall i.\, apply_{\sigma,\alpha}(sdf, [u_1, \ldots, u_k], v_i, r_i, t_i)\end{array}}{\sigma, \alpha \vdash \texttt{SUMIF}(e_0, e_1, \ldots, e_n) \Downarrow \sum\{v_i \mid r_i = 1 \wedge i = \{1, \ldots, n\}\}, 1 + c_0 + \sum_{j=1}^{n} c_j + \sum_{i=1}^{n} t_i} \text{ (sumif)}$$

$$\frac{\begin{array}{c}\sigma,\alpha \vdash e_1 \Downarrow FunVal(sdf,[u_1,\ldots,u_k]),c_1 \\ k = arity(sdf)-1 \qquad \sigma,\alpha \vdash e_2 \Downarrow Av(1,h,[[v_{i1}]]),c_2 \\ \sigma,\alpha \vdash e_3 \Downarrow v_3,c_3 \qquad v_3 \in Number \qquad w' = \lfloor v_3 \rfloor + 1 \\ \forall i = 1,\ldots,w'.\, apply_{\sigma,\alpha}(sdf,[u_1,\ldots,u_k],r_{i-1},r_i,t_i) \\ r_0 = [[v_{*1}]] \qquad \sigma,\alpha \vdash_v Av(w',h,[[r_0 : \ \ldots \ : r_n]]) \Downarrow arr,c_4\end{array}}{\sigma,\alpha \vdash \texttt{HSCAN}(e_1,e_2,e_3) \Downarrow arr, 1+c_1+c_2+c_3+c_4+\sum_{i=1}^{n} t_i} \text{ (hscan)}$$

Figure 8.6 – Cost semantics for a subset of Funcalc's higher-order built-in functions [6].

- Rule (map) is the rule for the intrinsic `MAP` function which is in fact a generalised n-ary `zip` function. Its first argument is expected to evaluate to a function value and successive arguments must be array values or cell areas. Given one such argument, `MAP` behaves like a regular map function. Given $n > 1$ arguments, `MAP` picks n element-wise values from the n arguments and passes them to the function value which is expected to have $arity(sdf) = n$.

  The rule's premises state that e_0 may evaluate to a function value with k early-bound arguments. Each of the other arguments $e_1,\ldots,e_n$ may evaluate to array values of equal size. The function is applied to each value in an element-wise fashion in the array value arguments $v_1,\ldots,v_n$ at each position (i,j). Each result r_{ij} is the elements of the array value in the conclusion at the corresponding position. The total cost is one plus the cost of evaluating the function value, the cost of evaluating all the array value arguments and the cost of all $i \cdot j$ function applications.

- The `COLMAP` function is similar to `MAP` but is not n-ary and maps column-wise as its name implies. A row-wise `ROWMAP` function also exists. Rule (colmap) states that e_1 may evaluate to a function value and e_2 may evaluate to an array value. The function value must have an arity equal to the height of the array value of e_2 since each column is mapped by the function. We use the new notation for referring to single columns. The call to `COLMAP` may evaluate to a new single-row, array value where each element is the result of a function application of each column in the input array value.

- The variadic `COUNTIF` function counts all the arguments which satisfy a predicate function. Rule (countif) states that if the first argument e_0 may evaluate to an unary function value and the remain-

ing expressions to some values $v_1, \ldots, v_n$, then a call to `COUNTIF` may evaluate to the number of arguments for which the result r_i of the function application is true (1).

- Rule (sumif) closely resembles rule (countif) but the conclusion may instead evaluate to a summation of the values v_i where $r_i = 1$ (true), i.e. the sum of values for which the predicate holds.
- Finally, we have the rule for `HSCAN`. The function performs a horizontal row-wise exclusive scan operation, as opposed to an element-wise scan as per Blelloch [93]. The rule is rather complicated so an example is in order. Consider the function sheet in figure 8.7a containing the definitions of functions `F` and `G`. The data sheet in figure 8.7b contains values 1 and 2 in cells A1 and A2. Suppose we call the following function as an array formula having selected the cell area A4:C5 to contain the result.

```
HSCAN(CLOSURE("F"), A1:A2, 2)
```

Each row in the result uses the first element in the corresponding row of the input array value as in a regular exclusive scan. The last argument to `HSCAN` (in this case 2) controls the number of additional columns produced by the call. In this call, the result will thus have three columns. The i^{th} column c_i is the result of applying the function `F` i times $\mathsf{F}^i(v_{i,j-1})$.

Let us return to the (hscan) rule. The premises say e_1 may evaluate to a function value accepting one argument, e_2 may evaluate to a single-column array value and e_3 may evaluate to a number. The result array *arr* may then evaluate to an array value where each column is the function applied $0, \ldots, n$ times to the input array value and then concatenated using the colon operator that we used for rule (hcat). Since a function applied zero times is the identity function, the first column is just the original input column which is why the number of columns in *arr* is $v_3 + 1$. The total cost is one plus the cost of evaluating the arguments, the cost c_4 of allocating a new array value for the result and sum of costs of the function applications.

A sensible implementation of `HSCAN` would avoid the quadratic work of multiple redundant function applications and use the results of previous columns. For example, applying the function

	A
1	=DEFINE("G", A3, A2)
2	
3	=A2+1
4	
5	=DEFINE("F", A7, A6)
6	
7	=MAP(CLOSURE("G"), A6)

(a)

	A	B	C
1	1		
2	2		
3			
4	1	2	3
5	2	3	4

(b)

Figure 8.7 – Column-wise scan using HSCAN. The horizontal scan is done on cell area A1:A2 and performed as an array formula in cell area A4:C5. Each value from the preceding column is incremented by one.

once at column i to the result of the previous column $i-1$ corresponds to having applied the function i times to the original input column.

8.2 Cost of Recalculation

With the addition of costs and the γ environment to our semantics, one may wish to know cost of a recalculation.

8.2.1 Minimal Recalculation

Let $dirty(ca)$ be the transitive closure of cell address ca in the support graph and every cell whose formula is volatile. That is, $dirty(ca)$ is the set of cells that need to be recalculated when ca is changed. The total cost of a minimal recalculation is thus the sum of costs of evaluating each cell in $dirty(ca)$.

$$minimalcost = \sum_{ca \in dirty(ca)} \gamma(ca)$$

8.2.2 Full Recalculation

A full recalculation evaluates all cells in the spreadsheet. These cells are exactly those in $dom(\phi)$.

$$fullcost = \sum_{ca \in dom(\phi)} \gamma(ca) + \sum_{ae \in dom(\alpha)} \gamma(ae)$$

8.3 Extended Consistency Requirements

We may also wish to extend the consistency requirements of section 2.9 to include costs.

$$dom(\sigma) = dom(\phi) \tag{8.2}$$
$$\forall ca \in dom(\phi).\ \sigma, \alpha \vdash \phi(ca) \Downarrow \sigma(ca), \gamma(ca) \tag{8.3}$$
$$\forall ae \in dom(\alpha).\ \sigma, \alpha \vdash ae \Downarrow \alpha(ae), \gamma(ae) \tag{8.4}$$
$$dom(\gamma) = dom(\phi) \cup dom(\alpha) \tag{8.5}$$

Requirement 8.2 remains unchanged. Requirements 8.3 and 8.4 now include costs via the γ environment. Requirement 8.5 is new and states the domain of the cost environment γ must be the union between the domains of ϕ and α as it contains the costs of both formulas and array formulas.

8.4 Implementation

To estimate the cost of a cell in Funcalc, we built a cost evaluator that applies the cost semantics. This was relatively straightforward as Funcalc already provides a suitable framework for building interpreters. The cost evaluator is thus simply an interpreter that returns a value and an associated concrete cost. We do not delve into the full details of the implementation of the cost evaluator here but instead refer readers to our technical report [6].

However, there is one particularly interesting part of the cost evaluator that is worth discussing. Since SDFs are compiled to CIL bytecode, how do we estimate their cost? We could develop another cost semantics for CIL instructions but we instead choose to directly interpret SDFs by evaluating the output cell and following dependencies back to the input cells. This is exactly what would have been done if there had been no SDF compiler. This raises yet another concern because we must decide how to load arguments for a call to an SDF and somehow handle recursive calls by proper abstraction of $\rho : Addr \to Value$, the local cell environment or stack frame of an SDF.

To implement ρ, we could directly modify the input cells of the SDF on each call but this would modify cells in the spreadsheet and need to be replaced when performing recursive calls. We have chosen instead

to keep track of an internal, local environment $l_{env} : Addr \rightarrow Value$ that mimics ρ and acts exactly like a stack frame. When an SDF is called, we create and push a new local environment onto an internal stack and store its parameters there by mapping the addresses of the input cells to their respective parameter values. This is identical to the semantic rule $(c5v)$ for function application where input parameters are stored in ρ' i.e. $\rho'(in_1) = v_1 \ldots \rho'(in_n) = v_n$. When the recursive call returns, we pop the top-most environment from the stack. Cost evaluation of a cell reference is modified to first look in the top-most local environment during a function call. If there is no such environment we examine the actual cells in the spreadsheet.

8.5 Cost Benchmarks

We ran the cost evaluator on the six LibreOffice Calc [44] spreadsheets and a subset of the EUSES corpus [28] to both see the range of costs of actual spreadsheets as well as the time taken to evaluate their cost. Table 8.1 contains these results. Costs correspond to applying the γ function to each cell address *ca* in the spreadsheet as in a full recalculation, but we exclude cells in the definition of SDFs.

At a glance, we notice that there seems to be no correlation between the number of formula cells and the time taken to evaluate the cost of each cell in the spreadsheet. This is to be expected as the formula count does not tell us anything about the complexity about each individual formula. For example, the `ny_emit99` and `Time` spreadsheets have almost the same number of formula cells but vastly different concrete and abstract costs and runtime.

For the LibreOffice Calc spreadsheets, it takes between half a minute to 50 minutes to evaluate the cost of the entire spreadsheet. As the cost evaluator is used to estimate the cost of cells in the static partitioning algorithm of chapter 9, it also affects the overall partitioning time as well.

8.6 Future Work

There are several interesting venues for future work to further develop the cost semantics.

Spreadsheet	Cost	Formulas	Runtime
LibreOffice Calc (runtime in *minutes*)			
building-design	978 520 000	108 332	0.56
energy-markets	2 175 001 469	534 507	50.20
grossprofit	4 423 203 701	135 073	38.74
ground-water	1 099 998 389	126 404	1.32
stock-history	1 230 276 358	226 503	1.42
stocks-price	1 165 235 199	812 693	22.41
EUSES (runtime in *milliseconds*)			
2004_PUBLIC_BUGS_INVENTORY	140 925	4495	28.83
Aggregate20Governanc#A8A51	723 436	3546	154.93
high_2003_belg	11 616 516	12 861	58.56
DNA	127 029	4715	15.76
EUSE	3463	413	1.27
PLANCK	25 200	806	13.33
02rise	91 581	10 316	26.64
financial-model-spreadsheet	20 128	3115	10.99
Financial-Projections	31 400	3649	11.04
2000_places_School	9286	1375	2.39
2002Qvols	10 222	2184	2.35
EducAge25	34 058	1470	6.19
notes5CMISB200SP04H2KEY	156 093	1557	103.60
Test20Station20Powe#A90F3	15 720	2164	5.59
v1tmp	6157	1129	2.06
MRP_Excel	415 529	4809	92.16
ny_emit99	76 010	4352	24.28
Time	33 832	4198	6.65
WasteCalendarCalculat#A843B	10 309	843	1.81
funding	280 702	1636	215.05
iste-cs-2003-modeling-sim	14 919	1991	6.71
modeling-3	1292	213	0.54

Table 8.1 – The total concrete cost, number of formula cells and the time taken to evaluate the cost of all cells in the LibreOffice Calc and EUSES spreadsheets. The cost evaluation was run 20 times and the average of those runs are shown in the fourth column. Note that the times for the LibreOffice Calc spreadsheets is shown in minutes and those for the EUSES spreadsheets in milliseconds.

8.6.1 Semantics As An Implementation Guide

Although Funcalc was developed before the cost semantics, the rules could be used for guiding other spreadsheet implementations by providing consistency and safety guarantees for both implementers and users. For example, they could be used to prove that optimisations preserve the behaviour of a computation and that they reduce the amount of work needed to perform a computation.

8.6.2 Off-loading Work to GPGPUs

LibreOffice Calc accelerated the recalculation of cell arrays using AMD general purpose graphics processing units (GPGPUs) [65]. In the same way that the cost semantics can guide a static partitioning algorithm, it could be used to estimate if there is a benefit to off-loading work to a GPGPU versus using sequential evaluation. This is critical to performance as the work-load has to be large enough to offset the cost of copying data to the GPGPU.

8.6.3 Type Systems for Spreadsheets

The semantics can pave the way for a type system for spreadsheets. Much work has already been done to introduce user-friendly type systems [80, 94–102]. See our technical report for more information on these systems [4]. In general, spreadsheets have very relaxed rules on types and employ heavy use of coercion. For example, Excel will coerce `TRUE` in the formula `=TRUE+1` to 1 giving us the result 2. Likewise, users expect the `SUM` function to coerce blank cells to 0 so that cells from disjoint cell areas can be summed by one call to `SUM`. From a programming language perspective, we would perhaps instead prefer that these cases resulted in a type error as they pose a risk. For example, if a blank cell in the summation later contains a value, this could lead to incorrect results. Type systems could afford additional safety guarantees to the spreadsheet paradigm to potentially avoid many common errors.

8.6.4 Formal Verification of Spreadsheets

Another potential use case for a semantics is as a foundation for static analysis and formal verification. For example, an end-user may wish to formally verify that certain critical properties of a computation hold such as lying within some numerical boundaries as spreadsheets are used to influence important business decisions. Another tool could identify performance bottlenecks in a spreadsheet and even suggest possible improvements.

8.6.5 Depth and Span

One future development to the cost semantics could include depth (also called span), the length of the longest sequential dependence in the

sense of Blelloch [93] for better estimation of parallelism. As mentioned at the beginning of this chapter, other work [91] has such a semantics that includes work and depth.

8.6.6 Abstract Interpretation

The big-step cost semantics is a *concrete* semantics that either produces a result value v at cost c or does not terminate because ordinary interpretation also does not terminate. An example would be a recursive SDF that never terminates. It is a first step towards a more general framework of abstract interpretation of spreadsheets based on ideas presented by Schmidt [103]. Our technical report [6] details preliminary work to move our concrete semantics to the abstract domain, some of which we briefly discuss here.

Besides termination, abstract interpretation has other benefits. First, it may be more accurate in some cases. Consider the following familiar expression and suppose that $\gamma(\mathrm{A1}) \ll \gamma(\mathrm{B1})$.

```
=IF(RAND()<0.5, A1, B1)
```

In the concrete semantics, we might under- or overestimate the cost of the expression based on which branch is evaluated. An abstract semantics may instead use some unification of the costs of the two branches to reflect that the true cost of the expression is not known until runtime. One such overestimation could be the maximum of the cost of the two branches i.e. $\max(\gamma(A1), \gamma(B1))$.

Due to the higher-order nature of Funcalc, preliminary work has also been done on a closure analysis [104] to improve cost estimates of higher-order function application.

Chapter 9

Static Partitioning

The contents of this chapter are mainly taken from the paper "*Static Partitioning of Spreadsheets for Parallel Execution*" [3].

The big-step cost semantics of chapter 8 is one half of a two-part cost model for estimating the cost of cells in the static partitioning algorithm that is the subject of this chapter. The dynamic, local approaches of chapters 5 and 7 attempted to accelerate *minimal* recalculation by discovering parallelism during cell evaluation. In contrast, static partitioning attempts to partition cells into groups and extract parallelism from the global structure of the spreadsheet before evaluating any cells. This corresponds to parallel *full* recalculation.

The static partitioning algorithm is inspired by work on an optimising compiler for the functional first-order programming language SISAL [105] due to the similarities between SISAL and Funcalc. To give readers some historical context, we give a brief historical account of SISAL in section 9.1. Afterwards, we discuss the second part of the cost model for estimating the synchronisation cost between different groups of cells when executing a partition in section 9.2. A formal definition of the partitioning problem in the context of spreadsheets is given in section 9.3. A preprocessing and postprocessing step is added to the algorithm in sections 9.4 and 9.5 respectively. The preprocessing step exploits the presence of cell arrays and the additional parallelism they expose, while the postprocessing step applies an optimisation to the partitioned spreadsheet. Section 9.6 describes three extensions to the algorithm to exploit parallelism in cell arrays. We present our results in section 9.7 and discuss them in section 9.8. The chapter is concluded by

a section on future work in section 9.9.

9.1 SISAL Background and Similarities to Funcalc

In chapter 3, we gave a brief historical account of dataflow and its various uses as well as its relation to spreadsheets. Here, we focus on coarse-grained dataflow and the work on the SISAL programming language [37, 105–107]. It was developed in the early 1980's to supersede Fortran as the primary language for scientific computing. The language was entirely side-effect free and adhered to the single-assignment rule: once a variable had been assigned a value, it could not later be reassigned. During program compilation, an intermediate acyclic graph format called IF1 [108] was generated which modelled program dataflow. The IF1 format was intended to be used by code generators to target different systems.

Sarkar worked on an optimising compiler that could automatically extract parallelism by analysing the IF1 graph. The compiler used a model of the hardware it was running on along with an execution profile of the given program. An approximate, iterative partitioning algorithm was used to partition the program. A partition consisted of a hierarchy of different node types. For example, compound nodes were used for control flow, parallel nodes for parallel iterative constructs such as map or reduce, and simple nodes for code that is executed sequentially. A compound node could e.g. contain a set of simple nodes to represent different outcomes of the control flow. Starting from some initial, fine partition, the algorithm iteratively merged groups as dictated by an objective function balancing the trade-off between the amount of synchronisation and parallelism in the partition. Merging continued until the coarsest possible partition was reached encompassing the entire program graph. The partition that minimised the objective function during merging was selected as the output of the algorithm. The partitioned program was then scheduled onto available processors at runtime.

Sarkar's system targeted both shared-memory multicore and distributed systems. SISAL programs were shown to run on par with Fortran programs on a Cray Y-MP/864 supercomputer [19] and thus helped dispel the, at the time general belief, that functional super-computing was impractical. This belief stemmed partly from the fact that early functional languages used copying for variable updates to retain the

single-assignment property [32] which was considered too inefficient for large-scale supercomputing.

Sarkar's algorithm was approximate due to the partitioning problem being NP-hard in the strong sense in general [37, 109]. This was shown by transformation of the 3-partition problem [109], which is NP-hard in the strong sense, to an instance of Sarkar's formulation of partitioning. Similarly, Wack [64] showed that all but a few special cases of the partitioning problem are NP-complete in the strong sense. However, Sarkar proved that his approximation algorithm yielded partitions within a factor of two from the optimal partition of a given program according to his objective cost function. For more information on SISAL and the history of dataflow languages in general, we encourage readers to consult [32] and [30].

We can now begin to see the aforementioned similarities between SISAL and Funcalc. Both are functional languages with the exception that Funcalc is higher-order so they share the properties of immutability and side-effect freedom. The intermediate IF1 graph format resembles the support graph of a spreadsheet. Recall that the support graph is similar to a dataflow graph where nodes are cells and information flows along edges from cells to their supported cells. The question we ask in this chapter is if we can transfer the ideas of SISAL's optimising compiler to the world of spreadsheets to accelerate full recalculation.

Figure 9.1 depicts the different stages of the static partitioning algorithm. We view the spreadsheet in terms of its support graph from which we generate some initial fine-grained partition of cells. The algorithm iteratively and greedily merges pairs of cell groups as dictated by a cost function that, as in Sarkar's work, balances the trade-off between parallelism and synchronisation overhead. Merging continues until the coarsest partition is reached, consisting of a single group of cells with no synchronisation overhead but no parallelism either. The partition that minimises the cost function is selected as the output of the algorithm and scheduled for parallel execution. We use the Task Parallel Library (TPL) to execute the partition. In the next section, we discuss the second part of the cost model for estimating the cost of synchronisation between groups of cells in a partition before giving a problem formulation of the partitioning algorithm.

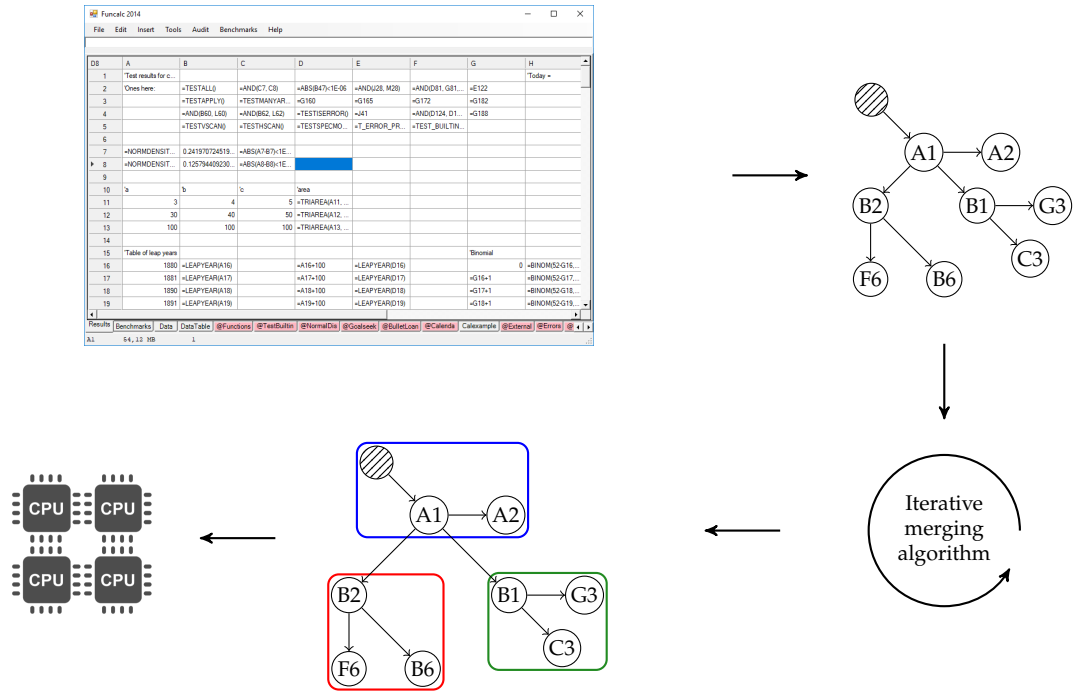


Figure 9.1 – Conceptualisation of static partitioning of spreadsheets. The spreadsheet is first viewed in terms of its support graph. An initial partition is generated. Then an iterative merging algorithm merges groups in partition until we reach a single group of cells that includes all cells in the spreadsheet. The best partition of the support graph consists of three groups of cells that can be scheduled to run on shared-memory multicore processors.

9.2 Synchronisation Cost

As mentioned earlier, Sarkar's system targeted both shared-memory multicore and distributed systems which called for a very general model that could accommodate different target systems. Our algorithm targets only shared-memory multicore architectures where the cost model must capture synchronization between threads.

Getting precise measurements for the cost of synchronisation is difficult in modern computers where several parts can affect communication between processors. Our test machine has three levels of cache per core. Caches are essential for the performance of contemporary computers as they minimise memory access latency. This severely complicates estimating the cost of synchronisation as a piece of memory may lie in different caches: a memory access may take anywhere from a few cycles to hundreds of cycles depending on how deep we need to transcend the cache hierarchy. This traversal also depends heavily on particular memory access patterns in the program, perhaps in parts we do not control directly such as synchronisation in the TPL. Cache coherency protocols [110] may also affect memory accesses. For example, synchronisation between threads may greatly affect performance by forcing the cache coherency protocol to invalidate cached memory to ensure consistency across processors.

Due to all these complications, we opted for a constant cost for synchronisation between threads. The constant was chosen partly based on benchmarks and experimentation. This may seem overly simplistic since a constant cannot take memory latency and other hardware aspects into account, but even this simplified model of communication is sufficient for generating partitions capable of accelerating spreadsheet computation as we shall see. We suggest possible improvements in section 9.9 to give a more precise model based on hardware and the operating system.

9.3 Problem Formulation

Having described the overall goal of partitioning and its associated cost model, we now formally define the problem of partitioning a spreadsheet. We view a spreadsheet in terms of its support graph $G = (V, E)$ consisting of a set of formula cell addresses $V = \{ca_1, \ldots, ca_n\}$ and a set of support edges $E \subseteq (V \times V)$. We exclude constant cells from V since

they involve no work. Let $G' = (V, E')$ be the dependency graph, the inversion of the support graph G. We wish to partition V into an acyclic partition $P_f = \{\tau_0, \ldots, \tau_m\}$ consisting of disjoint groups of cells $\tau_i \subseteq V$ minimising the objective function F. We require that all formula cells are contained in some group: $\bigcup_{i=0}^{m} \tau_i = V$. A partition P can be viewed as a condensation of G and we refer to this condensed graph as G_τ where vertices are the set of groups of cells in the partition $\{\tau \mid \tau \in P\}$. The forward and dependency edges of G_τ are dictated by the predecessor and successor sets we define shortly.

The partition P_f must minimise an objective function F: $\arg\min F(P) = P_f$ but we do not require P_f to be optimal. Furthermore, any partition P produced by the algorithm must be acyclic to ease scheduling but we defer a detailed discussion until section 9.3.4. We define the following functions on a group τ.

$$\mu : ca \to \tau \tag{9.1}$$

$$\textsc{Pred}(\tau) = \{\mu(ca_1) \mid \forall (ca_1, ca_2) \in E' \,.\, \mu(ca_1) \neq \tau \wedge \mu(ca_2) = \tau\} \tag{9.2}$$

$$\textsc{Succ}(\tau) = \{\mu(ca_2) \mid \forall (ca_1, ca_2) \in E \,.\, \mu(ca_1) = \tau \wedge \mu(ca_2) \neq \tau\} \tag{9.3}$$

$$\textsc{Time}(\tau) = \sum_{ca \in \tau} \gamma(ca) \tag{9.4}$$

$$\textsc{Sync}(\tau) = \text{Synchronisation cost of a } \tau \tag{9.5}$$

Given a partition, we can construct a mapping μ from each cell address to the τ it is assigned in the given partition. The set of predecessors Pred of a τ in G_τ is given by equation (9.2). We find all cell dependency edges in E' which start in τ and end in some different $\tau_i \neq \tau$. The definition of the successor set Succ in equation (9.3) is analogous. Equation (9.4) says the time required to execute a τ is the summation of the costs of all cell addresses in τ given by the cost environment γ we defined in chapter 8. Finally, the synchronisation cost of a τ is given by Sync in equation (9.5) which returns our synchronisation constant from section 9.2.

9.3.1 The Objective Function F

The objective function $F(P)$ approximates the trade-off between parallelism and synchronisation in a partition P and thus gives its cost or quality in regard to these two properties. It is used to guide partitioning

to produce a partition that maximises parallelism and minimises synchronisation cost. The objective function is the maximum of the *critical path* and *overhead* terms [37] given by equations (9.6) to (9.8).

$$\textsc{Sync}(P) = \sum_{\tau \in P} (|\textsc{Pred}(\tau)| + |\textsc{Succ}(\tau)|) \cdot \textsc{Sync}(\tau) \tag{9.6}$$

$$\textsc{Time}(P) = \sum_{\tau \in P} \textsc{Time}(\tau) \tag{9.7}$$

$$F(P) = \max\left(\frac{\text{CPL}(P)}{\textsc{Time}(P) \div N}, 1 + \frac{\textsc{Sync}(P)}{\textsc{Time}(P)}\right) \tag{9.8}$$

The total synchronisation cost of a partition P is given by equation (9.6). It is the number of predecessors and successors of a τ times its synchronisation cost. This models two-way synchronisation by including both predecessors and successors.

The total time to evaluate a partition P is given by equation (9.7) and is simply the summation of the time taken to execute each $\tau \in P$. Although a partition prescribes some distribution of cells into groups the amount of work in a partition remains constant.

Finally, the objective function in equation (9.8) is the maximum of the *critical path* and *overhead* terms, the two terms in the max function. The former term is the critical path length, the most expensive sequential path in G_τ, given by the CPL function, divided by the ideal parallel execution time of P on N total processors. The term is a ratio of how close a partition P is to the ideal speed-up. The overhead term is one plus the synchronisation cost of P normalised by the time taken to execute P. The addition of 1 ensures that the overhead term and F by extension will always be ≥ 1 to avoid a zero cost.

To illustrate how F balances synchronisation overhead and parallelism, consider the finest and coarsest granularity partitions. The former may expose ample parallelism but cause the overhead term to dominate F due to increased overall synchronisation. The latter may have little synchronisation due to lack of parallelism causing the critical path term to dominate F. The idea is that some intermediate partition between the two extreme granularities will balance the two terms and minimise F.

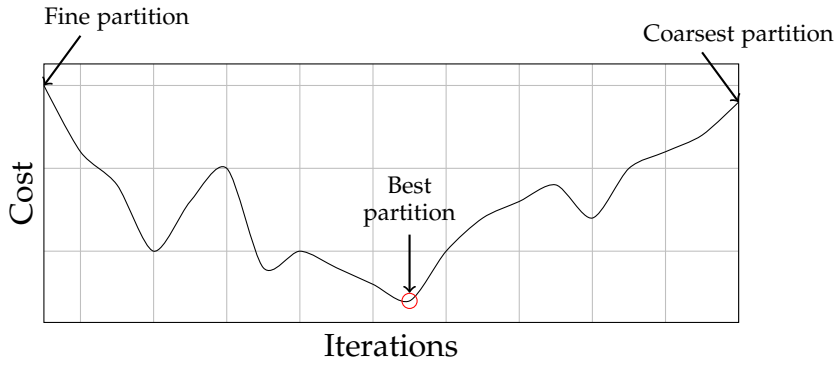


Figure 9.2 – Fictive cost evolution of the objective function during iterative merging. We start from some initial, fine partition and continue merging groups until we reach the coarsest partition. The idea is that the best partition that minimises the objective function lies somewhere in between these two extremes. It provides the balance between synchronisation cost and parallelism.

9.3.2 Iterative, Greedy Group Merging

Given the problem formulation of the previous section, we now describe the iterative, greedy and approximate merging algorithm that finds the ideal partition. We start by generating some initial fine granularity partition P_i. In Sarkar's work, P_i was generated using a lower-bound cost threshold. Nested groups in the hierarchical program graph were merged until they exceeded the threshold. Our work generates an initial partition from cell arrays but we defer a detailed discussion until section 9.4. For now it suffices to assume each cell is assigned a distinct τ in P_i. Next, we construct the PRED and SUCC sets for each τ and assign costs to each cell using our cost evaluator from chapter 8. By extension this also assigns costs to all groups in P_i. We compute the cost $F(P_i)$ of P_i and record it as the best cost obtained so far.

Starting from P_i, we iteratively and greedily merge pairs of groups (τ_1, τ_2) until we reach the coarsest partition consisting of a single τ containing all formula cells with no parallelism but no synchronisation overhead either. We select the intermediate partition that minimized F as the output P_f of the algorithm. The algorithm is greedy since it never backtracks and considers different possible merges in cases where a merge yields a partition of poor quality.

Which two groups τ_1 and τ_2 do we select for a merge at each itera-

tion? We select τ_1 as the group with the largest synchronisation cost in hopes of reducing the partition's overall synchronisation overhead [37]. We select τ_2 as the group that yields the smallest change in the critical path length if we were to merge it with τ_1 to retain as much parallelism in the new partition as possible. A fictive cost evolution as a function of iterations of the iterative merging process is depicted in figure 9.2.

9.3.3 Acyclic Constraint

We noted earlier that partitions must be acyclic to alleviate scheduling. To keep all partitions acyclic during merging, we impose an *acyclic constraint*[1] on each partition [37]. When two groups τ_1 and τ_2 are selected for a merge, we also merge any τ that lies on a path between τ_1 and τ_2, and thus outside the *convex subgraph* defined by τ_1 and τ_2. The precise definition of a convex subgraph is given in definition 2.

Definition 2. A subgraph H of a directed graph G is convex if for every pair of vertices $a, b \in H$, any path between a and b is fully contained in H.

For example, if there is a path $\tau_1 \to \tau_i \to \tau_2$ and we did not merge τ_i as well, we would introduce a cycle in G_τ. Figure 9.3 shows what the constraint intuitively means. It shows τ_1 and τ_2 that are about to be merged as denoted by the enclosing, red rectangle. Two other groups τ_i and τ_j exist. The constraint prohibits a τ from spawning and waiting for work so τ_j must be merged. It also prohibits fork-join parallelism where the fork happens at τ_1 and the join at τ_2. Consequently, τ_i must also be merged. While this may remove some parallelism from the partition, it greatly simplifies scheduling which is the subject of the next section.

9.3.4 Scheduling Partitions

Since the output of the merging algorithm P_f is acyclic, we can schedule it by first topologically sorting G_τ by its dependencies and create a TPL [71] task to run each τ. Iterating through the topologically sorted list, we either (1) mark a τ without any dependencies, i.e. $\textsc{Pred}(\tau) = \varnothing$, as a *source* and create a task to evaluate it; or (2) create a TPL *continuation* task. This special task waits for all its dependent tasks to finish before

[1] Originally referred to as the *convexity constraint* in [37] as it relates to convex subgraphs.

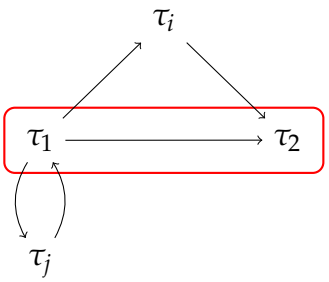


Figure 9.3 – The two cell groups τ_1 and τ_2 have been selected for a merge. The acyclic constraint prohibits a τ from spawning work, such as between τ_1 and τ_j, and fork-join parallelism, such as that between τ_1, τ_i and τ_2.

starting, analogous to the firing rule of dataflow. Every source task is started and we wait for all tasks to complete. The cells in each τ are thus evaluated sequentially. Only the execution of each τ in G_τ is done in parallel if possible.

How do we handle cyclic dependencies? Although G_τ is guaranteed to be acyclic, each τ may still contain cells that have cyclic dependencies. Each non-source task first checks if all its dependent tasks ran to normal completion before starting execution. If not, it does not start evaluating its cells and instead propagates any errors to its successors so that execution can quickly terminate. Source tasks detect cyclic dependencies as normal (see section 4.1) and propagate this information to the remaining tasks.

One may be wary of the use of the TPL given the performance debugging of chapter 6 but the static partitioning algorithm was developed before we were aware of the causes of the performance issues. For the task-based parallel cell interpreter, spawning a task for each cell was too fine a granularity that overburdened the garbage collector. However, the partitioning algorithm produces a much coarser distribution of work and spawns overall less tasks. This will hopefully both ensure a work distribution with a more suitable task granularity and less overhead for the garbage collector. It is however possible that a very fine granularity partition is selected as the best one which will still spawn many tasks. Even so, we may still get decent speed-ups for some spreadsheets as we saw with the task-based interpreter.

Next, we discuss a preprocessing and a postprocessing step of the partitioning algorithm. The preprocessing step generates an initial par-

	A	B
1	1	=R[+0]C[-1]*2
2	2	=R[+0]C[-1]*2
3	3	=R[+0]C[-1]*2

Figure 9.4 – A cell array with data-parallel, row-wise independent computations on the cells in A1:A3. References are given in the R1C1 reference format.

tition that takes advantage of the parallelism in cell arrays. The postprocessing step optimises sequential dependency chains in the final partition P_f.

9.4 Cell Array Preprocessing

Most spreadsheets are highly structured and contain cell arrays that describe bulk, possibly data-parallel, operations on a collection of cells. In figure 9.4, the cell array in cells B1:B3 essentially maps the values in cells A1:A3 by multiplying each value by two. Each row in the cell array could be computed in parallel since there are no transitive references among its cells that refer back into the cell array. To be able to exploit this parallelism when scheduling a partition, we include a preprocessing step that assigns each cell array to its own τ in the initial partition P_i. This creates a much coarser initial partition than assigning each cell to its own τ which lowers the overall partitioning time as less cells need to be considered for a merge. Non-cell array cells are still assigned to a distinct τ.

The preprocessing step also lets us optimise a different part of the algorithm's initialisation, namely constructing the predecessor and successor sets of each τ. Normally, we examine the dependencies and support sets of all cells in each $\tau \in P_i$. For a large number of cells, this can be a time-consuming process relative to the partitioning itself. In this section, we introduce an *optimistic* cell array analysis to deduce the predecessor and successor sets by taking advantage of the fact that some groups contain only a cell array. Therefore, the analysis *only* concerns τ's that are assigned cell arrays in P_i.

The main idea of the analysis derives from the shared formula expressions in cell arrays, potentially letting us examine only a few cells in each cell array instead of all its cells. For example, consider the two single-column cell arrays in figure 9.5 which could depict the spread-

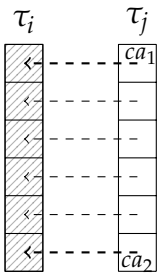


Figure 9.5 – Two cell arrays belonging to τ_i and τ_j. The cells in τ_j's cell array refer to cells in τ_i.

sheet in figure 9.4. It would suffice to know that the top cell ca_1 and bottom cell ca_2 of the cell array in τ_j referenced the top and bottom cells of the cell array in τ_i. Due to the shared formula expressions, the four middle cells in τ_j also refer to cells in τ_i and we can conclude that $\tau_i \in \textsc{Pred}(\tau_j)$ and conversely $\tau_j \in \textsc{Succ}(\tau_i)$. If the cell arrays instead contained thousands of cells the analysis would save a lot of time.

The analysis is optimistic as it assumes it will succeed in most cases. More complicated scenarios may cause the analysis to fail in which case we fall back to examining each individual cell. As the analysis only needs to examine a few cells before either succeeding or failing, its impact on the overall partitioning time should be negligible. Cell arrays are common in spreadsheets [24] so the analysis should in general be beneficial. Having established some intuition for the analysis, we proceed to formalise it in the next section.

9.4.1 Formal Cell Array Analysis

We first formally define the cell array analysis in a naive way where each cell in the cell array is examined then incrementally improve it to examine fewer cells. To simplify the explanation of the analysis, we consider only single-cell references, and disregard cell area reference and array formulas. We conclude the analysis in section 9.4.3 by showing a special case where a cell reference can effectively be considered absolute, and how the analysis handles cell area references and array formulas.

Formally, a cell array is described by a cell area $ca_1 : ca_2$ where equation (9.9) holds for all pairs of cell addresses in the cell array.

$$\forall ca_i, ca_j \in ca_1 : ca_2 \,.\, \phi(ca_i) = \phi(ca_j) \tag{9.9}$$

(a) Single cell. **(b)** Single row. **(c)** Single column. **(d)** Cell area.

Figure 9.6 – The different types of cell areas a single-cell cell reference can refer to from a cell array.

I.e. all cells in the cell array share the same formula expression given by the $\phi : Addr \rightarrow Expr$ function we originally defined in section 2.8. To query the dependencies of all cells in a cell array we could define a function Ω, as in equation (9.10), to collectively query all cells in the cell array and pass the cell area $ca_1 : ca_2$, containing the cell array, to Ω. The function returns the cell area that is referenced by the cell array: a rectangular region, single-row or single-column region or just a single cell, depending on the relativity and absoluteness of the cell references in the shared formula of the cell array and its shape.

$$\Omega(area, cr) = area' \tag{9.10}$$

This idea is shown in figure 9.6. A fully absolute cell reference such as `=$A$2` would mean that all cells refer to a single cell (figure 9.6a). A relative and absolute cell reference such as `=A$2` or `=$A2` would mean that a single-row or single-column area may be referenced, respectively (figures 9.6b and 9.6c). A fully relative cell reference such as `=A2` would mean the cells may reference a cell area (figure 9.6d).

Instead, we use a function ω to query individual cells before consulting Ω. Equations (9.11) and (9.12) define the ω and δ functions. We distinguish between a cell address *ca* such as B2 or G5 that denotes a position in the spreadsheet and a cell reference *cr* such as A1, R[+0]C[-1] or A1:B2 that denote a cell reference in an expression.

$$\omega : (ca, cr) \rightarrow ca' \tag{9.11}$$

$$\delta : ca \rightarrow \{\text{all cell references } cr_i \text{ in the expression } \phi(ca)\} \tag{9.12}$$

The ω function takes a cell address *ca* and a cell reference *cr* and returns a new cell address ca' by evaluating the cell reference *cr* in the context of *ca*. For example, evaluating the expression `=R[+0]C[-1]` in

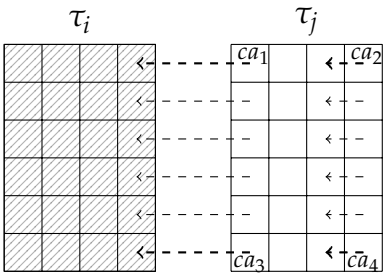


Figure 9.7 – Two cell arrays where the left-most cells in the cell array on the right refer to cells in the cell array on the left. The remaining cell references in the cell array of τ_j are transitive.

cell B1 would refer to cell A1. Finally, the δ function returns an *ordered* set of cell references contained in the expression $\phi(ca)$. We will later show why the set must be ordered.

We could now construct the predecessor set of a τ as in equation (9.13) by looking at the dependencies of each cell in the cell array similarly to using Ω. Recall that the function $\mu : ca \to \tau$ accepts a cell address and returns the τ to which it belongs. The question now is what subset of cells we should examine with ω to avoid examining every cell.

$$\textsc{Pred}(\tau) = \{\mu(\omega(ca, cr)) \mid cr \in \delta(ca), ca \in \tau\} \tag{9.13}$$

In figure 9.7, it would suffice to examine just cell addresses ca_1 and ca_3 of τ_j's cell array since all other references are either transitive or refer to cells in τ_i. Recall that a cell in a cell array is transitive if it refers to a cell in the cell array itself, and intransitive if it only refers to cells outside the cell array. Transitive cell references can be disregarded so we can ignore ca_2 and ca_4 in figure 9.7. We query only the four unique corner cells of a cell array since they delimit the extremities of its dependencies. We specify *unique* corner cells since they coincide for single-row and single-column cell arrays. If the remaining cells agree on their τ predecessors, we conclude that all the cells in the cell array can reach that predecessor. In figure 9.7, the referenced cells of ca_1 and ca_3 both belong to τ_i: $\mu(\omega(ca_1, cr)) = \mu(\omega(ca_3, cr)) = \tau_i$ and so we conclude that $\tau_i \in \textsc{Pred}(\tau_j) \wedge \tau_j \in \textsc{Succ}(\tau_i)$.

Not all cell arrays contain such simple dependencies. A more complicated scenario is depicted in figure 9.8. We once again ignore the transitive references of ca_2 and ca_4 but now $\mu(\omega(ca_1, cr)) = \tau_i$ and

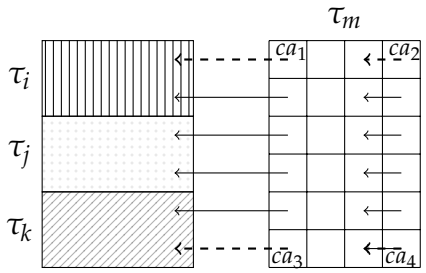


Figure 9.8 – A trickier case where examining only the corner cells of the cell array of τ_m would miss its dependency on τ_j.

$\mu(\omega(ca_3, cr)) = \tau_k$ in the cell array of τ_m. Thus we cannot conclude anything about the dependencies of the remaining cells in the cell array of τ_m and must fall back to querying all cells to avoid missing τ_m's dependency on τ_j.

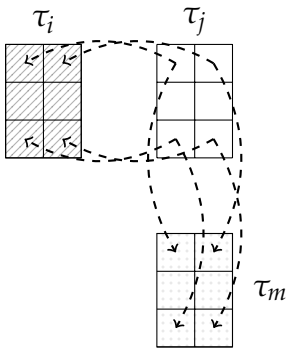
Let $C(\tau)$ be a function for retrieving the unique corner cells of a τ. Then we can redefine the predecessor set given in equation (9.13) for a τ containing its cell array by only examining its corner cells.

$$\textsc{Pred}(\tau) = \{\mu(\omega(ca, cr)) \mid cr \in \delta(ca), ca \in C(\tau)\} \tag{9.14}$$

Until now, we have given several definitions but not the actual criteria for a successful analysis. The analysis succeeds for a τ if there is no pair of corner cell addresses $(ca_i, ca_j) \in C \times C \wedge ca_i \neq ca_j$ where we consider each ordered pairwise intransitive cell reference $(cr_i, cr_j) \in (\delta(ca_i), \delta(ca_j))$ and find that they do not agree on the set of predecessors they refer to. We can consider cell references in the expressions of the corner cells in a pairwise manner since they share a formula expression and δ returns them as an ordered set. The conditions for a successful analysis are given by equation (9.15) for a τ whose cell array spans the cell area $ca_1 : ca_2$.

$$\begin{aligned} &\forall (ca_i, ca_j) \in C(\tau) \times C(\tau) \wedge ca_i \neq ca_j, \\ &\forall (cr_i, cr_j) \in (\delta(ca_i), \delta(ca_j)) \\ &\qquad \text{where } \omega(ca_i, cr_i) \notin ca_1 : ca_2 \wedge \omega(ca_j, cr_j) \notin ca_1 : ca_2\,. \\ &\qquad \mu(\omega(ca_i, cr_i)) \setminus \mu(\omega(ca_j, cr_j)) = \varnothing \end{aligned} \tag{9.15}$$

The condition is best illustrated by the example in figure 9.9 where the cell array of τ_j depends on the cell arrays of τ_i and τ_m. Cell addresses



(a)

	A	B	C	D	E
1	=SIN(F1)	=SIN(G1)		=A1+D5	=B1+E5
2	=SIN(F2)	=SIN(G2)		=A2+D6	=B2+E6
3	=SIN(F3)	=SIN(G3)		=A3+D7	=B3+E7
4					
5				=COS(F5)	=COS(G5)
6				=COS(F6)	=COS(G6)
7				=COS(F7)	=COS(G7)

(b)

Figure 9.9 – A more complicated relationship between three cell arrays which shows when the cell array analysis succeeds. It also shows why we must consider cell references in the same order to avoid drawing incorrect conclusions about reachability between the cell arrays.

ca_1 (D1) and ca_3 (D3) both refer to cells in τ_i using the first cell reference in their expressions (A1 and A3). Similarly, ca_3 (D3) and ca_4 (E3) refer to cells in τ_m using the second cell reference (D7 and E7). The remaining combination of pairs of cell addresses and cell references also refer to the same τ. We see now why we require δ return an *ordered* set of cell references. If we considered different cell references in different parts of the expression at two distinct cells, say cell reference A1 at ca_1 and D7 at ca_3, we would incorrectly conclude that one could reach a cell in τ_i and another a cell in τ_m.

9.4.2 Pseudo-Absolute Cells

Even cell references that are not absolute in both dimensions can be considered absolute in the context of a cell array as shown in figure 9.10. Since the cell array in column B refers to cell A1 using a row-absolute but

	A	B
1	=PI()	
2		
3	6	=2*A$1*A3
4	2	=2*A$1*A4
5	...	...

Figure 9.10 – Spreadsheet calculating the circumference $2\pi r$ of various circle radii. Cell A1 holds the constant π which the cell array in column B refers to. Since the reference is row-absolute and column-relative, all cells in the cell array always refer to A1.

column-relative reference, all cells in the cell array will always refer to that cell. The same is true for row-relative, column-absolute references and single-row cell arrays. This scenario occurs in the `building-design` spreadsheet.

9.4.3 Cell Area References and Array Formulas

Handling cell area references and array formulas is straightforward but added additional complexity to the explanation of the cell array analysis so we discuss them separately in this section.

For cell area references, we require that the entirety of the referenced area lies within a single τ. If it does not, we cannot know if cells outside the referenced τ belong to another group of cells so we fall back to checking all cells in the cell area.

Array formulas are not technically cell arrays but the principles of the analysis can still be applied. The cells of an array formula are elements from an array generated by a single formula expression, so we can simply apply the analysis to the array formula by considering it as a cell array with a single cell.

9.5 Postprocessing Sequential Dependencies

The approximate algorithm is not guaranteed to produce an optimal partition [37] and may miss obvious optimisations. One such case is a sequential chain of dependencies in G_τ that are assigned to different τ's in the final partition. Much like the thread-local evaluation of the task-based interpreter in chapter 5, we could avoid unnecessary synchronisation by instead assigning the entire chain to a single τ. Therefore, we

traverse G_τ to find sequential chains and ensure that they are assigned to a single τ in the final partition P_f.

9.6 Extensions

As mentioned earlier in section 9.3.4 on scheduling partitions, cells within each τ are evaluated sequentially but independent τ's may be executed in parallel. This disregards any additional parallelism inside each τ. The cell array preprocessing step assigns all cell arrays to their own τ so some groups in the final partition may contain data-parallel formula expressions. In this section, we present three extensions to the algorithm to exploit this additional parallelism: the first extension uses *nested parallelism* within cell arrays; the second extension uses an instance of our parallel task-based cell interpreter from chapter 5 to evaluate each τ; the third extension uses a tool [22] to rewrite cell arrays to calls to SDFs that can be evaluated in parallel.

9.6.1 Nested Cell Array Parallelism Extension

The TPL was designed so each thread in the threadpool has its own local queue in addition to a shared global work queue [83]. Each thread will first look for work in its local queue followed by the global queue. They ultimately resort to stealing work from other threads if they still have not found any work.

The nested parallelism extension relies on the fact that spawning tasks nested within another task enqueues the nested tasks in the current threadpool thread's local queue, circumventing the global queue and reducing contention. However, we cannot necessarily spawn a task for each cell in the cell array since its references may be transitive [22]. In figure 9.11, each cell reference in the cell array in column B refers to a cell five rows below it. Blindly spawning tasks for each cell would cause data races as there is no proper synchronisation between cell dependencies in the cell array. However, there is still some degree of parallelism we can exploit. In this instance, we can subdivide the transitive cell array into subgroups of five cells then evaluate each subdivision in parallel in a lock-step fashion [22]. Each subdivided group will not contain any transitive references to themselves. This requires additional analysis of the references of a cell array to determine if and how they can be

	A	B
1	1	=R[+5]C[+0]*2
2	2	=R[+5]C[+0]*2
3	3	=R[+5]C[+0]*2

Figure 9.11 – A transitive, single-column cell array in column B where each cell refers to a cell five rows below it. By subdividing the cell array into groups of five, we can still extract some degree of parallelism from it.

evaluated in parallel in this manner, but this analysis and the subdivision strategy for transitive cell arrays was not implemented due to time constraints.

If a cell array is intransitive, we can easily spawn a task for each cell in the cell array. The cells in a τ are only evaluated when all their inputs have been evaluated, so the dependencies of the cell array are already computed due to topological sorting during scheduling.

9.6.2 Task-Based Cell Interpreter Extension

The second extension uses an instance of our task-based parallel cell interpreter within each τ. Since the interpreter already provides proper synchronisation, we can safely use an instance of it inside each τ regardless of its contents and call `Eval` on each cell within a τ. Unlike the previous extension, we do not require an additional analysis of cell arrays since the algorithm already ensures proper synchronisation.

The interpreter follows the support graph by default in search of cells to compute but this would mean that cells in the successor set of a τ might be evaluated prematurely, violating that a τ is only evaluated when its dependencies are complete. To avoid this, we disallow the interpreter from following support edges by setting the `UseSupportSets` flag to false and do not use the global work queue.

Unfortunately, we implemented this extension before being aware of the data race in the task-based algorithm. However, since none of the spreadsheets contain non-deterministic functions, the race never manifests. In future work, we should use the thread-based interpreter instead.

	A	B
1	1	=R[-1]C[+0]+1*2
2	2	=R[-1]C[+0]+1*2
3	3	=R[-1]C[+0]+1*2

(a)

	A	B
1	1	{=MAP(CLOSURE("GEN_0"), A1:A3)}
2	2	{=MAP(CLOSURE("GEN_0"), A1:A3)}
3	3	{=MAP(CLOSURE("GEN_0"), A1:A3)}

(b)

Figure 9.12 – Rewriting a cell array to an array formula containing a call to Funcalc's intrinsic `MAP`. The `GEN_0` function is auto-generated by the system to capture the shared formula expression of the cell array.

9.6.3 Cell Rewriting Extension

Biermann et al. [22] analysed cell arrays and rewrote eligible ones to an array formula consisting of a call to an SDF. The analysis examined the patterns exhibited by the cell array's shared formulas, and a formula expression might be rewritten to a `map` or `prefix` operation or not rewritten at all if the shared expression is an unsupported pattern. Custom SDFs are generated if no built-in function already supports the computation of the shared formula. Replacing interpreted cells with calls to compiled functions led to good speed-ups even without parallelism, and even for the EUSES spreadsheets [28] that contain little computation. The spreadsheet is rewritten after being loaded from disk so no change to the static partitioning algorithm is necessary since we already handle array formulas. In the spectrum of automatic parallelism, this approach is global since it rewrites all eligible cell arrays in the spreadsheet and static because rewriting happens at load-time and not during cell evaluation.

The cell array in figure 9.12a essentially expresses a `map` operation which is rewritten to the array formula in figure 9.12b using a call to Funcalc's intrinsic `MAP`. The `GEN_0` function is an auto-generated SDF that captures the computation of the shared formula of the cell array. Biermann et al. parallelised rewritten cell arrays via the TPL but otherwise disjoint rewritten cell arrays were evaluated sequentially. Combining their method with our static partitioning algorithm parallelises both the rewritten cell arrays internally and evaluates them in parallel.

9.7 Results

In this section, we present the results of the base implementation of the static partitioning algorithm without any extensions and the results of

Spreadsheet	Cell Arrays	% of Formulas	Average Size	Rewritten
building-design	6	99.93%	18 042	6/0
energy-markets	76	99.99%	7032	76/0
grossprofit	9	99.94%	15 000	9/0
ground-water	12	100%	10 533	12/0
stock-history	22	99.97%	10 292	20/0
stocks-price	8	99.99%	101 578	8/0

Table 9.1 – Columns from left to right: The number of cell arrays in the LibreOffice Calc spreadsheets; the percentage of formulas contained in cell arrays; the average size of cell arrays; and the number of rewritten intransitive and transitive cell arrays. No transitive cell arrays are rewritten because none of the spreadsheets contain any.

the three extensions we proposed in section 9.6. Bear in mind that the static partitioning algorithm and these results were developed before we were aware of the cause of the performance issues we presented in chapter 6. Hence, some of the results and the ensuing explanations follow some of the observations we made for the task-based interpreter that also uses the TPL for parallel evaluation. This also means that the benchmarks were made with the LibreOffice Calc spreadsheets that contain variadic function calls to `AVERAGE`.

Figures 9.14 to 9.17 show speed-ups for running the final partition using the base implementation and its three extensions. The partitioning times are shown in figure 9.13. This is the time for taken for the iterative merging algorithm of section 9.3.2 to partition a spreadsheet after costs have been assigned. Table 9.2 lists the running times in seconds for all experiments.

9.7.1 Experimental Setup

The experimental setup is largely the same as for the dynamic algorithms of part I. One exception is that partitioning depends on the number of processors on the system due to the critical path term of the objective function, so it was necessary to generate partitions for each number of processors used in the experiments.

Furthermore, we did not test the static partitioning algorithm on the synthetic spreadsheets because they were purposefully designed to contain specific, artificial topologies that the algorithm would likely not be well-suited to partition.

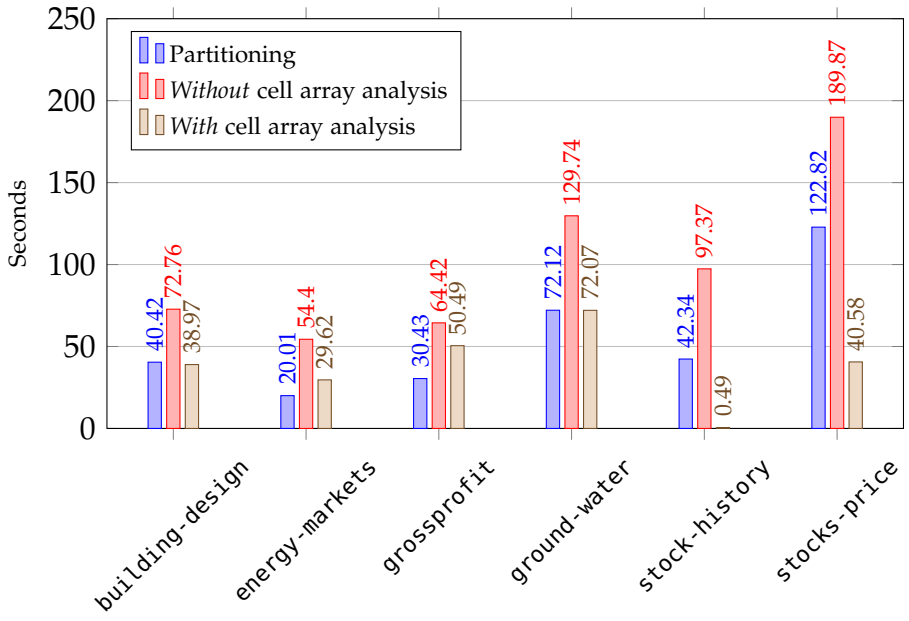


Figure 9.13 – Time in seconds to partition the spreadsheets using the iterative merging algorithm; and time in seconds to construct the predecessor and successor sets with and without the cell array analysis discussed in section 9.4. The values are averages of 10 runs.

9.8 Discussion

There are five key observations to be made from our results.

Observation 1 The LibreOffice Calc spreadsheets contain large cell arrays that contain almost all formula cells.

Table 9.1 shows that all our benchmark spreadsheets are dominated by large, intransitive cell arrays which contain almost all formula cells. There is no transitive cell arrays in any of the spreadsheets. The spreadsheets were quite possibly used to demonstrate using OpenCL kernels on AMD GPGPUs [65] to accelerate spreadsheet recalculation. This likely explains why they contain so many large intransitive, data-parallel cell arrays which are ideal for execution on GPGPUs. Observation 1 has two implications. First, there is a lot of parallel computation we can exploit with the three proposed extensions from section 9.6. Second, the preprocessing step successfully analyses most of the cell arrays since they are relatively simple. This is clearly evident from figure 9.13 where

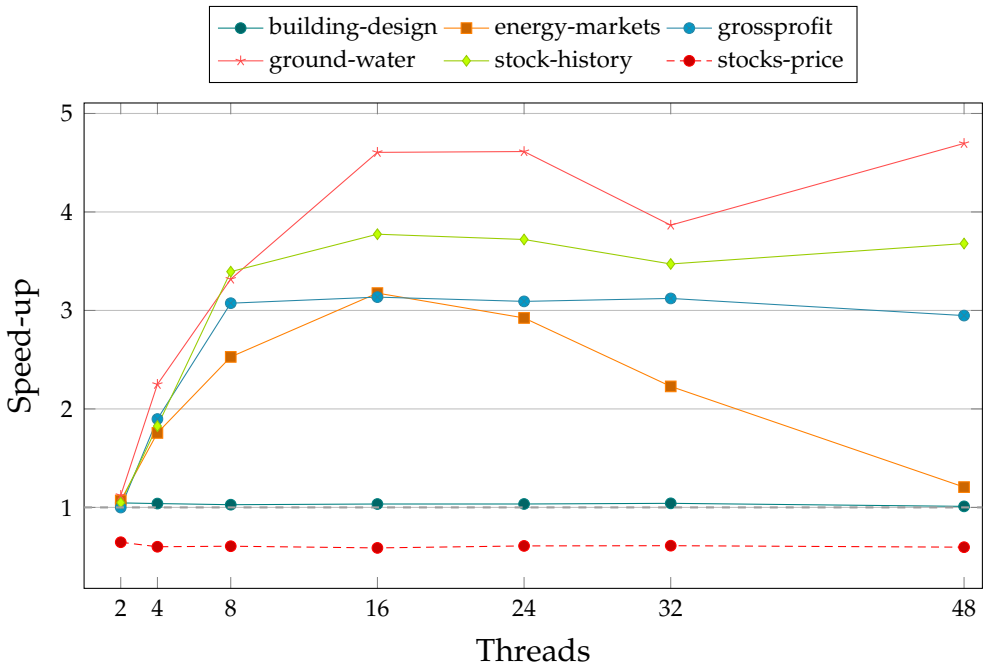


Figure 9.14 – Speed-ups over sequential performance for 20 runs of the LibreOffice Calc spreadsheets with the base implementation of the static partitioning algorithm. The grey, dashed line indicates the sequential baseline.

the cell array analysis has a beneficial impact on all spreadsheets. This is especially true for the `stock-history` spreadsheet where it takes around 97 seconds to construct the predecessor and successor sets without the analysis, and just around half a second using the analysis.

Partitioning itself, excluding cost assignment to cells, currently takes between 20 to 120 seconds as shown in figure 9.13. Overall, the entire process takes on the order of several minutes to almost an hour where the dominating factor is assigning costs to each cell (see table 8.1 on page 152). This is too long in our opinion and future work should focus on improving the speed of the cost evaluator.

Observation 2 Comparing the performance of the base implementation to its three extensions shows that it is necessary to exploit the internal parallelism of cell arrays.

We get mostly unimpressive overall speed-ups for the base implementation in figure 9.14 and table 9.2. When comparing these results to

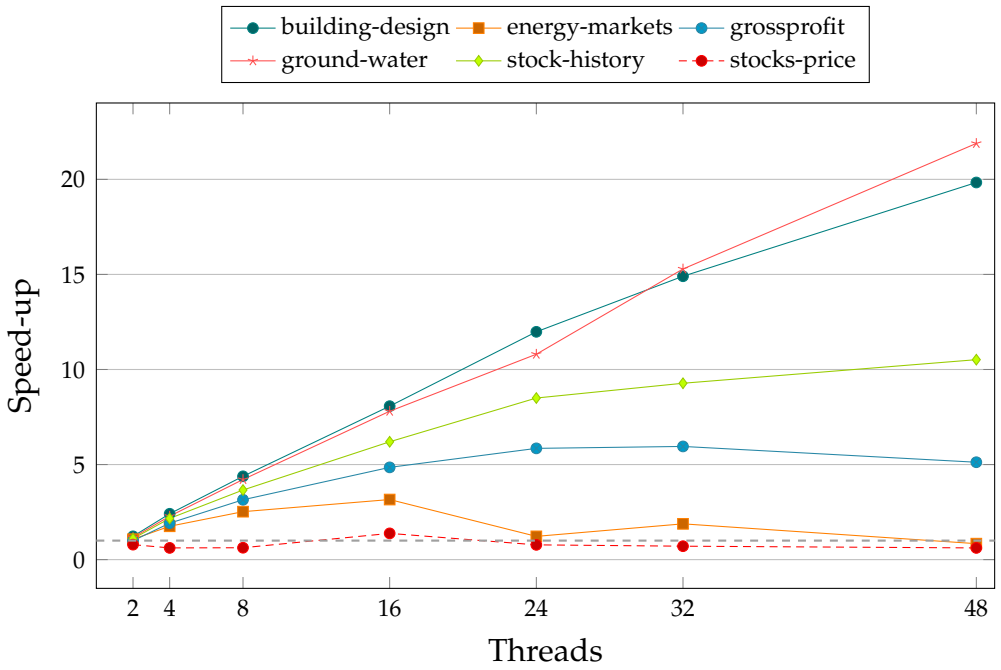


Figure 9.15 – Speed-ups over sequential performance for 20 runs of the LibreOffice Calc spreadsheets with the nested cell array parallelism extension of the static partitioning algorithm. The grey, dashed line indicates the sequential baseline.

those of the three extensions, it is evident that we must exploit the additional parallelism exposed by cell arrays to achieve better speed-ups. For spreadsheets with significantly less cell arrays and many non-cell arrays cells, the base implementation may still give decent speed-ups however.

Observation 3 The `energy-markets`, `grossprofit` and `stocks-price` spreadsheets have less predictable speed-ups and performance consistently peaks at 16 or 32 cores. Adding more cores seems to slow down recalculation.

We saw the same behaviour for the task-based interpreter in chapter 5 which also uses the TPL for parallelisation. This observation applies to the base implementation and all three extensions with the exception of `stocks-price` for the base implementation where the best speed-up is achieved at 2 cores, although it is still slower than sequential execution.

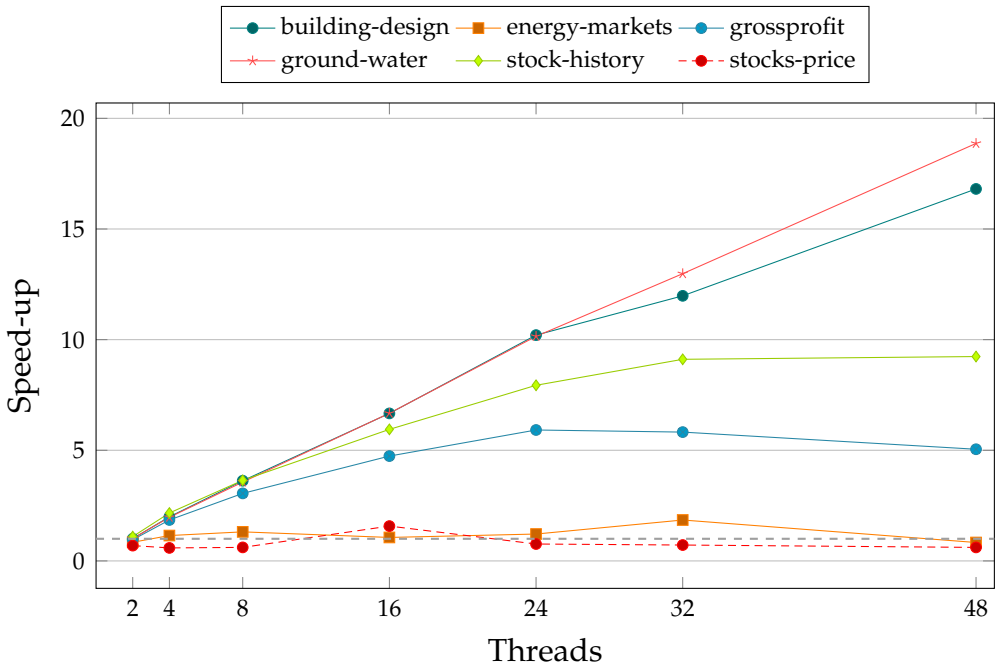


Figure 9.16 – Speed-ups over sequential performance for 20 runs of the LibreOffice Calc spreadsheets with the task-based parallel cell interpreter extension of the static partitioning algorithm. The grey, dashed line indicates the sequential baseline.

We still get approximately 1.3–3.0-fold speed-up for 16 and 32 cores for these spreadsheets which may be consistent with increased cross-chip communication in our Xeon machine.

One would also be inclined to suspect our simplified communication model which may cause poor merging decisions that lead to poorly partitioned spreadsheets. However, we still get good speed-ups for half of the spreadsheets, and furthermore the same divergence in performance was observed for the dynamic, parallel interpreters.

The presence of large, dominating cell arrays in all the spreadsheets means that most formula cells are initially assigned a τ as part of a cell array. Consequently, there was little variation in how the spreadsheets were partitioned for a varying number of processors and fine-grained partitions were generally favoured because merging often removed too much parallelism. Where partitioning varied, it was mostly the case that the τ's with cell arrays were merged together for a smaller number of processors to minimise the overhead term of the objective func-

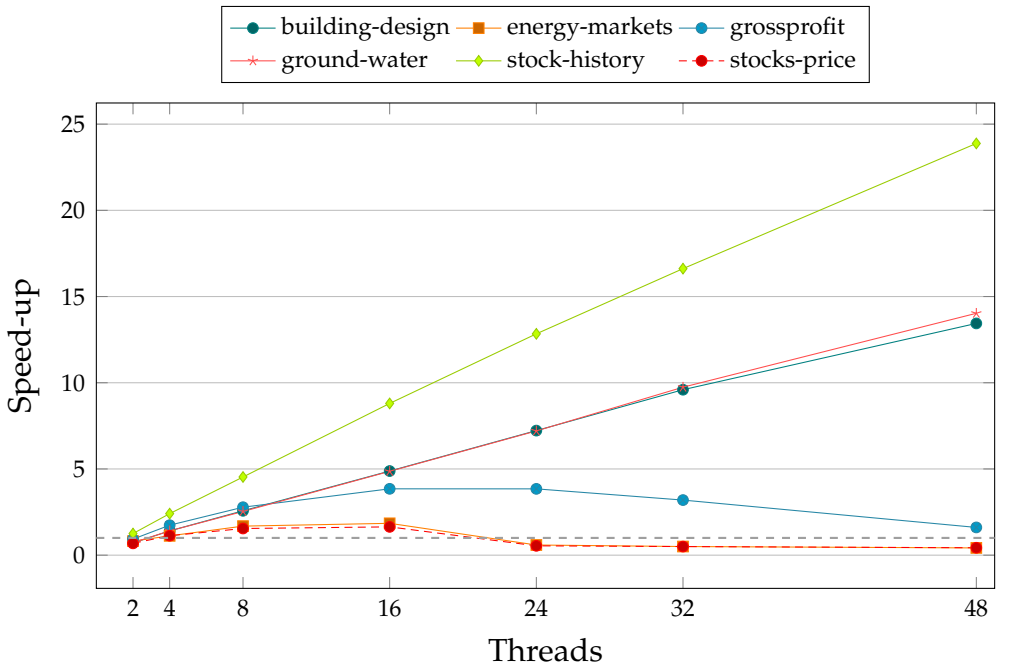


Figure 9.17 – Speed-ups over sequential performance for 20 runs of the LibreOffice Calc spreadsheets with the cell rewriting extension of the static partitioning algorithm. The grey, dashed line indicates the sequential baseline.

tion. In this case, not much parallelism can be exploited anyway for a small number of cores. For the `building-design`, `ground-water` and `stock-history` spreadsheets, this level of granularity leads to good speed-ups.

Observation 4 The nested cell array extension produces the best overall speed-ups on 48 cores.

Out of the three extensions, the nested cell array parallelism extension gives the overall best speed-ups on 48 cores with a maximum 21.89-fold speed-up for the `ground-water` spreadsheet.

Observation 5 The cell rewriting extension achieves different speed-ups compared to the other extensions for some spreadsheets while the nested cell array and task-based cell interpreter extensions achieve similar speed-ups.

Spreadsheet	Number of Threads 1	2	4	8	16	24	32	48
Base Implementation								
building-design	32.12	**30.72**	30.92	31.26	31.05	31.05	30.85	31.79
energy-markets	168.16	157.08	95.75	66.51	**52.95**	57.52	75.45	139.41
grossprofit	102.19	102.33	53.86	33.25	**32.59**	33.05	32.73	34.66
ground-water	81.26	72.42	36.13	24.49	17.65	17.61	21.02	**17.30**
stock-history	64.90	61.90	35.54	19.12	**17.20**	17.44	18.69	17.64
stocks-price	102.74	**158.94**	171.10	169.56	174.53	168.72	168.10	172.34
Nested Cell Array Parallelism Extension (section 9.6.1)								
building-design	32.12	26.23	13.32	7.33	3.98	2.68	2.16	**1.62**
energy-markets	168.16	156.68	95.84	66.67	**53.22**	137.52	89.35	200.28
grossprofit	102.19	102.72	53.31	32.46	21.06	17.46	**17.16**	19.95
ground-water	81.26	69.97	35.59	19.29	10.41	7.52	5.32	**3.71**
stock-history	64.90	58.84	29.94	17.73	10.47	7.64	7.00	**6.17**
stocks-price	102.74	130.46	166.70	164.37	**74.44**	132.28	145.48	166.87
Task-based Parallel Cell Interpreter Extension (section 9.6.2)								
building-design	32.12	31.97	16.12	8.86	4.82	3.15	2.68	**1.91**
energy-markets	168.16	199.63	146.66	128.06	158.50	138.43	**90.95**	202.04
grossprofit	102.19	106.70	55.22	33.48	21.57	17.27	**17.55**	20.25
ground-water	81.26	80.47	41.22	22.81	12.17	8.01	6.26	**4.31**
stock-history	64.90	59.05	29.97	17.81	10.92	8.18	7.12	**7.03**
stocks-price	102.74	148.56	174.75	168.01	**65.26**	134.47	143.08	168.25
Cell Rewriting Extension (section 9.6.3)								
building-design	32.12	45.35	22.79	12.49	6.58	4.45	3.35	**2.39**
energy-markets	168.16	206.16	150.10	99.84	**91.04**	285.22	335.60	400.26
grossprofit	102.19	109.75	58.79	36.74	**26.56**	26.57	31.98	63.51
ground-water	81.26	111.90	57.74	32.07	16.71	11.28	8.34	**5.79**
stock-history	64.90	51.81	26.94	14.31	7.37	5.06	3.91	**2.72**
stocks-price	102.74	149.98	91.09	66.45	**62.60**	189.57	205.99	239.11

Table 9.2 – Evaluation time in seconds for different number of cores for the base implementation and its three extensions. Bold numbers denote the fastest runs for each spreadsheet. The standard deviation was within ± 0.08 for all results, except for the base implementation running `stocks-price` on 16 cores with a standard deviation of ± 0.18.

Table 9.1 shows that all intransitive cell arrays are rewritten except for two in the `stock-history` spreadsheet and that no transitive cell arrays exist in any of the spreadsheets. Consequently, the majority of formula cells are evaluated via compiled code instead of interpretation. Interestingly, this leads to different results than the other two extensions. For example, the `stock-history` spreadsheet now runs in 2.72 seconds compared to 6.17 and 7.03 seconds for the other two extensions, yielding the best speed-up out of all the results with a 23.88-fold speed-up on 48 cores. We believe more efficient and parallelisable SDFs may be generated for the `stock-history` spreadsheet. The `building-design` and `ground-water` spreadsheets have gone from around 20-fold speed-ups to just below 15-fold speed-up. The `energy-markets` and `stocks-price`

spreadsheets have even worse performance on 48 cores however but their peak performance at 16 and 32 cores is comparable to the peak performances of the other two extensions. We hoped that rewritten cell arrays calling compiled code would yield much better speed-ups.

The two other extensions achieve similar speed-ups since their method of parallelisation is similar. The nested cell array extension has slightly better performance than the task-based parallel interpreter extension. This is likely because it spawns tasks for each cell in the cell array without the need for additional synchronisation. The task-based interpreter extension is built to evaluate any kind of topology, not just cell arrays, so it must use additional synchronisation for claiming cells and setting their state although dependencies will already have been evaluated due to the topological sorting during scheduling.

9.9 Future Work

We could rectify this by caching and reusing the computed costs if cells are not modified between partitioning but cost assignment would still need to be done at least once in full. However, it is possible to save the partition in the file when the spreadsheet is written to disk so we can quickly load it next time without having to partition again.

9.9.1 Improved Cost Model

There is more work to be done on our cost model. In particular, the synchronisation costs should more closely reflect the system hardware. For example, we could run benchmarks to better estimate the synchronisation costs of the system as done by Sarkar [37] who generated execution profiles of the program. The system's cache hierarchy and number of physical processor chips could also be taken into account. E.g. the depth of the cache hierarchy may affect the average memory access time and separate physical chips may incur off-chip synchronisation. Sarkar's machine model was quite low level and detailed and e.g. modelled the costs of reads, writes and delays of the system. We do not believe such a low-level model is necessary to achieve good speed-ups as already evident from our results. Wack [64] ran experiments on different networked systems to construct a cost model to reflect communication costs of the system. Alternatively, for GPU acceleration in LibreOffice Calc the cost

of a cell array is determined via weights [66] based in part on the complexity of the formula expressions. This is much faster than using a cost evaluator based on a big-step cost semantics but may be very imprecise. The weights may be sufficient for generating good partitions that can accelerate recalculation. No information is given on how these weights are generated or if they can take different hardware architectures into account.

The cost evaluator currently uses unit costs but one could instead substitute them for timing values generated from benchmarks to increase precision. Overall, the performance of the cost evaluator was not satisfactory. In the worst case, it takes around 50 minutes for the `energy-markets` spreadsheet which is not acceptable given the poor speed-ups we achieve for half of the LibreOffice Calc spreadsheets. Future work should thus strive to improve the performance of the cost evaluator.

9.9.2 Chunked Cell Array Parallelism

We developed the static partitioning algorithm before we were aware of the performance issues related to fine-grained task spawning as we presented in chapter 6. Consequently, we spawn a task for each cell in each cell array leading to similar, albeit slightly better, performance profiles. Instead, we could minimise the impact of fine-grained task spawning by chunking the cell array into a number of equally sized portions corresponding to the number of processors in the system. This spawns coarser granularity tasks with more work which may yield better overall speed-up than simply spawning a task per cell in each cell array.

9.9.3 Preprocessing Sequential Dependencies

The postprocessing step of section 9.5 merged τ's in sequential dependency chains in G_τ to avoid unnecessary synchronisation. This was primarily done to exploit an obvious optimisation that would sometimes not be achieved using the iterative merging algorithm alone. However, one could also apply this postprocessing step as a preprocessing step instead in order to merge such sequential dependencies in the beginning of the algorithm instead, lowering the overall partitioning time. We are unsure if such long sequential dependencies occur in practice however.

Part III

Future Work and Conclusion

Chapter 10

Future Work

In the final part of the dissertation, we summarise directions for future work that in our opinion should be prioritised by future researchers.

10.1 Continued Performance Debugging

In chapter 6, we presented our efforts to better understand the performance issues of the task-based interpreter of chapter 5. This lead to an improved thread-based interpreter in chapter 7 that yielded better speed-ups and removed the race condition we discovered in our implementation of speculative reevaluation. However, even the improved speed-ups still yielded only between 3- and 5-fold speed-ups on some of the sheets for 48 logical cores, despite that the time spent garbage collection had decreased significantly for all spreadsheets. Furthermore, this performance divergence recurred in all our results of all our algorithms. This strongly suggests that there is perhaps further causes for the poor performance in the implementation of Funcalc that should be resolved before the development of new parallel algorithms.

Further instrumentation of both the sequential and parallel implementations using different profiling tools, than those used in chapter 6, can perhaps give us deeper insights. For example, there are a plethora of commercial profiling tools we could use to investigate false-sharing and cache misses which were not supported as performance counters by the WPM on our system.

10.2 Diversity of Benchmark Spreadsheets

The LibreOffice Calc and synthetic spreadsheets all had useful properties for benchmarking. The former contained large cell arrays for the static partitioning algorithm to exploit, and the latter helped verify whether our dynamic algorithms were agnostic to the topology of a spreadsheet. However, future work should strive to use even more diverse spreadsheets.

For example, it would be interesting to run our algorithms on large spreadsheets with little to no cell arrays. This would result in more fine-grained initial partitions and consequently require more merging steps to partition, increasing the overall partitioning time. Such spreadsheets could help further develop the static partitioning algorithm to better handle spreadsheets without the presence of large cell arrays.

It would also be interesting to benchmark spreadsheets that contain many user-defined VBA functions that are part of computation in the spreadsheet. None of the LibreOffice Calc spreadsheets contained VBA functions that were part of the actual computation in the spreadsheet. VBA functions can be slow and such spreadsheets would allow us to convert such functions to SDFs and more clearly show their performance benefits over VBA functions.

10.3 Combining the Dynamic and Static Algorithms

The dynamic, local interpreters and the static partitioning algorithm are not mutually exclusive and could be combined. During the development of a spreadsheet, a user would update parts of the spreadsheet, automatically using the dynamic interpreter for minimal recalculation. Once the user is satisfied with the spreadsheet, the user could enable static partitioning to partition the spreadsheet. Users can change the inputs to the spreadsheet program and immediately use the optimised partitioned spreadsheet to accelerate recalculation. Since the structure of the spreadsheet remains the same when changing its inputs, we can readily reuse the partition. In cases where the structure of the spreadsheet does change, one could imagine performing incremental updates to the partition. Currently, we repartition the entire spreadsheet if a cell is modified. Although such incremental updates are likely not trivial to implement, they would undoubtedly be very useful for end-users.

Chapter 11

Conclusion

In this dissertation, we have investigated different approaches to automatic parallelism in an effort to accelerate spreadsheet recalculation for end-user development.

Chapters 5 and 7 explored dynamic algorithms that exploit local parallelism. We showed how both algorithms could achieve good speed-ups on a set of benchmark spreadsheets. In chapter 8, we developed a cost semantics for Funcalc to estimate the cost of evaluating a cell. To the best of our knowledge, this is the first such semantics for a spreadsheet and we suggested many interesting use cases for the semantics besides cost estimation. We developed a cost evaluator to apply the cost semantics to a cell, which was used in the development of a static algorithm that partitions a spreadsheet to extract global parallelism and schedules the partitioned spreadsheet onto shared-memory multicore processors.

We believe that our work has shown that it is feasible to implement automatic parallelism in spreadsheets to the benefit of end-users. The algorithms require little to no interaction from end-users and transparently leverage the shared-memory multicore processors of the system. SDFs give end-users more expressive power in the form of compiled, higher-order functions that are defined in a paradigm that end-users already understand and are familiar with. This work is therefore a first step towards a powerful spreadsheet framework for end-user development. We hope it will pave the way for a paradigm shift where spreadsheets are viewed as a serious computational tool by "real" programmers for a broad range of complex and computationally demanding problems.

Bibliography

[1] Alexander Asp Bock and Florian Biermann. "Puncalc: task-based parallelism and speculative reevaluation in spreadsheets". In: *The Journal of Supercomputing* (Mar. 23, 2019). ISSN: 1573-0484. DOI: 10.1007/s11227-019-02823-8. URL: https://doi.org/10.1007/s11227-019-02823-8.

[2] Florian Biermann and Alexander Asp Bock. "Puncalc: Task-Based Parallelism and Speculative Reevaluation in Spreadsheets". In: *International Symposia on High-Level Parallel Programming and Applications* (July 2018).

[3] Alexander Asp Bock. "Static Partitioning of Spreadsheets for Parallel Execution". In: *Practical Aspects of Declarative Languages*. Ed. by José Júlio Alferes and Moa Johansson. Springer International Publishing, 2019, pp. 221–237. ISBN: 978-3-030-05998-9.

[4] Alexander Asp Bock. *A Literature Review of Spreadsheet Technology*. Technical report TR-2016-199. IT University of Copenhagen, Nov. 30, 2016. 33 pp. URL: http://forskningsdatabasen.dk/en/catalog/2350168960.

[5] Christopher Scaffidi, Mary Shaw, and Brad Myers. "Estimating the numbers of end users and end user programmers". In: *2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. 2005, pp. 207–214.

[6] Alexander Asp Bock, Thomas Bøgholm, Peter Sestoft, Bent Thomsen, and Lone Leth Thomsen. *Concrete and Abstract Cost Semantics for Spreadsheets*. Technical report TR-2018-203. Denmark: IT University of Copenhagen and AAlborg University, 2018. 56 pp.

[7] Alexander Asp Bock. *A Comparison Between SISAL 1.2 and Funcalc*. Technical report TR-2019-205. IT University of Copenhagen, 2018.

[8] Dan Bricklin and Bob Frankston. *VisiCalc: Information from its creators, Dan Bricklin and Bob Frankston*. URL: http://www.bricklin.com/visicalc.htm (visited on 06/16/2016).

[9] A. G. Yoder and D. L. Cohn. "Real spreadsheets for real programmers". In: *Proceedings of the 1994 International Conference on Computer Languages*. May 1994, pp. 20–30. DOI: 10.1109/ICCL.1994.288396.

[10] Rommert J. Casimir. "Real Programmers Don't Use Spreadsheets". In: *SIGPLAN Not.* 27.6 (June 1992), pp. 10–16. ISSN: 0362-1340. DOI: 10.1145/130981.130982. URL: http://doi.acm.org/10.1145/130981.130982.

[11] David Wile. "Lessons learned from real DSL experiments". In: *Science of Computer Programming* 51.3 (2004), pp. 265–290. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2003.12.006. URL: http://www.sciencedirect.com/science/article/pii/S0167642304000310.

[12] Chris Scaffidi. *Counts and earnings of end-user developers*. Sept. 21, 2017. URL: https://www.linkedin.com/pulse/counts-earnings-end-user-developers-chris-scaffidi?published=t (visited on 10/19/2017).

[13] David Wakeling. "Spreadsheet functional programming". In: *Journal of Functional Programming* 17.1 (2007), pp. 131–143. DOI: 10.1017/S0956796806006186.

[14] Robin Abraham, Margaret Burnett, and Martin Erwig. "Spreadsheet Programming". In: *Wiley Encyclopedia of Computer Science and Engineering*. American Cancer Society, 2009, pp. 2804–2810. ISBN: 9780470050118. DOI: 10.1002/9780470050118.ecse415. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470050118.ecse415.

[15] Mani Chandy. "Concurrent Programming for the Masses (Invited Address)". In: *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing*. PODC '85. Minaki, Ontario, Canada: ACM, 1985, pp. 1–12. ISBN: 0-89791-168-7. DOI: 10.1145/323596.323597. URL: http://doi.acm.org/10.1145/323596.323597.

[16] Herb Sutter. "The free lunch is over: A fundamental turn toward concurrency in software". In: *Dr. Dobb's journal* 30.3 (2005), pp. 202–210.

[17] Peter J. Denning and Jack B. Dennis. "The Resurgence of Parallelism". In: *Commun. ACM* 53.6 (June 2010), pp. 30–32. ISSN: 0001-0782. DOI: 10.1145/1743546.1743560. URL: http://doi.acm.org/10.1145/1743546.1743560.

[18] John Backus. "Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs". In: *Commun. ACM* 21.8 (Aug. 1978), pp. 613–641. DOI: 10.1145/359576.359579. URL: http://doi.acm.org/10.1145/359576.359579.

[19] David Cann. "Retire Fortran? A debate rekindled". In: *Communications of the ACM* 35.8 (Aug. 1992), pp. 81–89.

[20] Peter Sestoft. *Popular Parallel Programming (P3)*. 2014. URL: http://www.itu.dk/people/sestoft/p3/ (visited on 04/30/2017).

[21] Alaaeddin Swidan, Felienne Hermans, and Ruben Koesoemowidjojo. "Improving the Performance of a Large Scale Spreadsheet: A Case Study". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Osaka, Suita, Japan: IEEE, Mar. 2016, pp. 673–677. ISBN: 978-1-5090-1855-0. DOI: 10.1109/saner.2016.100. URL: http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2016-003.pdf.

[22] Florian Biermann, Wensheng Dou, and Peter Sestoft. "Rewriting High-Level Spreadsheet Structures into Higher-Order Functional Programs". In: *Practical Aspects of Declarative Languages*. Ed. by Francesco Calimeri, Kevin Hamlen, and Nicola Leone. Vol. 10702. Lecture Notes in Computer Science. Springer International Publishing, 2018, pp. 20–35. DOI: 10.1007/978-3-319-73305-0_2. URL: http://dx.doi.org/10.1007/978-3-319-73305-0%5C_2.

[23] Peter Sestoft. *Spreadsheet Implementation Technology*. The MIT Press, 2014. ISBN: 9780262526647.

[24] Wensheng Dou, Shing-Chi Cheung, and Jun Wei. "Is Spreadsheet Ambiguity Harmful? Detecting and Repairing Spreadsheet Smells due to Ambiguous Computation". In: *36th International Conference on Software Engineering (ICSE 2014)*. ACM. 2014, pp. 848–858.

[25] Wensheng Dou, Chang Xu, Shing-Chi Cheung, and Jun Wei. "CACheck: Detecting and Repairing Cell Arrays in Spreadsheets". In: *IEEE Transactions on Software Engineering (TSE)* (2016).

[26] Roland Mittermeir and Markus Clermont. "Finding high-level structures in spreadsheet programs". In: Feb. 2002, pp. 221–232. ISBN: 0-7695-1799-4.

[27] Robin Abraham and Martin Erwig. "Inferring Templates from Spreadsheets". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. New York, NY, USA: ACM, 2006, pp. 182–191. ISBN: 1-59593-375-1. DOI: 10.1145/1134285.1134312. URL: http://doi.acm.org/10.1145/1134285.1134312.

[28] Marc Fisher and Gregg Rothermel. "The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanisms". In: *SIGSOFT Softw. Eng. Notes* 30.4 (May 2005), pp. 1–5. ISSN: 0163-5948. DOI: 10.1145/1082983.1083242. URL: http://doi.acm.org/10.1145/1082983.1083242.

[29] Felienne Hermans and Emerson Murphy-Hill. "Enron's Spreadsheets and Related Emails: A Dataset and Analysis". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 2*. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 7–16. URL: http://dl.acm.org/citation.cfm?id=2819009.2819013.

[30] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. "Advances in Dataflow Programming Languages". In: *ACM Comput. Surv.* 36.1 (Mar. 2004), pp. 1–34. ISSN: 0360-0300. DOI: 10.1145/1013208.1013209. URL: http://doi.acm.org/10.1145/1013208.1013209.

[31] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. "Build systems a la carte". In: *Proc. International Conference on Functional Programming (ICFP'18)*. ACM, Sept. 2018. URL: https://www.microsoft.com/en-us/research/publication/build-systems-la-carte/.

[32] Paul G. Whiting and Robert S. V. Pascoe. "A History of Data-Flow Languages". In: *IEEE Ann. Hist. Comput.* 16.4 (Dec. 1994), pp. 38–59. ISSN: 1058-6180. DOI: 10.1109/85.329757. URL: https://doi.org/10.1109/85.329757.

[33] Arvind and R.S. Nikhil. "Executing a program on the MIT tagged-token dataflow architecture". In: *IEEE Transactions on Computers* 39.3 (Mar. 1990), pp. 300–318. ISSN: 0018-9340. DOI: 10.1109/12.48862.

[34] C.H. van Berkel, M.B. Josephs, and S.M. Nowick. "Applications of asynchronous circuits". In: *Proceedings of the IEEE* 87.2 (Feb. 1999), pp. 223–233. ISSN: 0018-9219. DOI: 10.1109/5.740016.

[35] Vivek Sarkar and John Hennessy. "Compile-time Partitioning and Scheduling of Parallel Programs". In: *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*. SIGPLAN '86. New York, NY, USA: ACM, 1986, pp. 17–26. ISBN: 0-89791-197-0. DOI: 10.1145/12276.13313. URL: http://doi.acm.org/10.1145/12276.13313.

[36] Vivek Sarkar and David Cann. "POSC–a Partitioning and Optimizing SISAL Compiler". In: *SIGARCH Comput. Archit. News* 18.3b (June 1990), pp. 148–164. ISSN: 0163-5964. DOI: 10.1145/255129.255152. URL: http://doi.acm.org/10.1145/255129.255152.

[37] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Research Monographs In Parallel and Distributed Computing. Cambridge, Massachusetts: MIT Press, 1989. ISBN: 0262691302.

[38] Gilles Kahn. "The semantics of a simple language for parallel programming". In: *In Information Processing '74: Proceedings of the IFIP Congress*. Vol. 74. 1974, pp. 471–475.

[39] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. "The synchronous data flow programming language LUSTRE". In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1305–1320. ISSN: 0018-9219. DOI: 10.1109/5.97300.

[40] Microsoft. *Dataflow (Task Parallel Library)*. URL: https://msdn.microsoft.com/en-us/library/hh228603(v=vs.110).aspx (visited on 05/06/2017).

[41] D. J. Power. *A Brief History of Spreadsheets*. Version 3.6. Aug. 30, 2004. URL: http://www.dssresources.com/history/sshistory.html (visited on 06/21/2016).

[42] Margaret Burnett, John Atwood, Rebecca Walpole Djang, James Reichwein, Herkimer Gottfried, and Sherry Yang. "Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm". In: *Journal of Functional Programming* 11.2 (2001), pp. 155–206.

[43] Google Inc. *Google Spreadsheets*. URL: https://www.google.com/sheets/about/ (visited on 06/02/2016).

[44] The Document Foundation. *LibreOffice Calc*. URL: https://www.libreoffice.org/discover/calc/ (visited on 05/09/2016).

[45] The GNOME Project. *Gnumeric*. URL: http://www.gnumeric.org/ (visited on 06/02/2016).

[46] Simon Peyton-Jones, Alan Blackwell, and Margaret Burnett. "A User-centred Approach to Functions in Excel". In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ICFP '03. New York, NY, USA: ACM, 2003, pp. 165–176. ISBN: 1-58113-756-7. DOI: 10.1145/944705.944721. URL: http://doi.acm.org/10.1145/944705.944721.

[47] Lee Benfield. "FMD: Functional Development in Excel". In: *Proceedings of the 2009 Video Workshop on Commercial Users of Functional Programming: Functional Programming As a Means, Not an End*. CUFP '09. New York, NY, USA: ACM, 2009. ISBN: 978-1-60558-943-5. DOI: 10.1145/1668113.1668121. URL: http://doi.acm.org/10.1145/1668113.1668121.

[48] Walter A.C.A.J. de Hoon, Luc M.W.J. Rutten, and Marko C.J.D van Eekelen. "Implementing a Functional Spreadsheet in Clean". In: *Journal of Functional Programming* 5 (1995), pp. 383–414.

[49] Rinus Plasmeijer, Marko van Eekelen, and John van Groningen. *Clean Language Report*. Version 2.2. Dec. 2011.

[50] Björn Lisper and Johan Malmström. "Haxcel: A Spreadsheet Interface to Haskell". In: *14th Int. Workshop on the Implementation of Functional Languages*. Forthcoming, 2002, pp. 206–222.

[51] Kurt W. Piersol. "Object-oriented Spreadsheets: The Analytic Spreadsheet Package". In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. OOPLSA '86. New York, NY, USA: ACM, 1986, pp. 385–390. ISBN: 0-89791-204-7. DOI: 10.1145/28697.28737. URL: http://doi.acm.org/10.1145/28697.28737.

[52] Matt McCutchen, Shachar Itzhaky, and Daniel Jackson. "Object Spreadsheets: A New Computational Model for End-user Development of Data-centric Web Applications". In: *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2016. Amsterdam, Netherlands: ACM, 2016, pp. 112–127. ISBN: 978-1-4503-4076-2. DOI: 10.1145/2986012.2986018.

[53] Florian Biermann and Peter Sestoft. "Quad Ropes: Immutable, Declarative Arrays with Parallelizable Operations". In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY 2017. Barcelona, Spain: ACM, 2017, pp. 1–8. ISBN: 978-1-4503-5069-3. DOI: 10.1145/3091966.3091971. URL: http://doi.acm.org/10.1145/3091966.3091971.

[54] R. A. Finkel and J. L. Bentley. "Quad trees: a data structure for retrieval on composite keys". In: *Acta Informatica* 4.1 (Mar. 1, 1974), pp. 1–9. ISSN: 1432-0525. DOI: 10.1007/BF00288933. URL: https://doi.org/10.1007/BF00288933.

[55] Hans-J. Boehm, Russ Atkinson, and Michael Plass. "Ropes: an Alternative to Strings". In: *Software – Practice and Experience* 25.12 (Dec. 1995), pp. 1315–1330.

[56] Peter Sestoft. "Corecalc and Funcalc Spreadsheet Technology in C#". In: (2014). URL: http://www.itu.dk/people/sestoft/funcalc/ (visited on 06/01/2016).

[57] Florian Biermann. *Declarative Parallel Programming in Spreadsheet End-User Development: A Literature Review*. Technical report TR-2016-192. IT University of Copenhagen, Feb. 2016.

[58] Felienne Hermans, Martin Pinzger, and Arie van Deursen. "Supporting Professional Spreadsheet Users by Generating Leveled Dataflow Diagrams". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. New York, NY, USA: ACM, 2011, pp. 451–460. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985855. URL: http://doi.acm.org/10.1145/1985793.1985855.

[59] Felienne Hermans, Martin Pinzger, and Arie van Deursen. "Breviz: Visualizing spreadsheets using dataflow diagrams". In: *Computing Research Repository* (2011). URL: http://arxiv.org/abs/1111.6895.

[60] Felienne Hermans, Ben Sedee, Martin Pinzger, and Arie van Deursen. "Data Clone Detection and Visualization in Spreadsheets". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 292–301. ISBN: 978-1-4673-3076-3. URL: http://dl.acm.org/citation.cfm?id=2486788.2486827.

[61] David Abramson, Paul Roe, Lew Kotler, and Dinelli Mather. "ActiveSheets: Super-computing with spreadsheets". In: *2001 High Performance Computing Symposium (HPC '01), Advanced Simulation Technologies Conference*. Citeseer. 2001, pp. 22–26.

[62] D. Abramson, R. Sosic, J. Giddy, and B. Hall. "Nimrod: a tool for performing parametrised simulations using distributed workstations". In: *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing*. Aug. 1995, pp. 112–121. DOI: 10.1109/HPDC.1995.518701.

[63] Microsoft. *HPC Services For Excel*. URL: https://technet.microsoft.com/en-us/library/ff877820(v=ws.10).aspx (visited on 06/30/2016).

[64] Andrew P. Wack. "Partitioning Dependency Graphs for Concurrent Execution: A Parallel Spreadsheet on a Realistically Modeled Message Passing Environment". PhD thesis. Newark, DE, USA, 1996. URL: http://portal.acm.org/citation.cfm?id=269551.

[65] Jim Trudeau. *Collaboration and Open Source at AMD: LibreOffice*. July 15, 2015. URL: https://developer.amd.com/collaboration-and-open-source-at-amd-libreoffice/ (visited on 07/31/2015).

[66] Michael Meeks. *Calc: The challenges of scalable arithmetic. Presented at FOSDOM*. 2018.

[67] Thomas Bøgholm, Kim G. Larsen, Marco Muniz, Bent Thomsen, and Lone Leth Thomsen. "Analyzing spreadsheets for parallel execution via model checking". In: *Essays on the Occasion of Bernhard Steffen's 60th Birthday (Lecture Notes in Computer Science, vol. 11200)*. Springer-Verlag, 2018.

[68] Nichlas Korgaard Møller. "Pre-Analyses Dependency Scheduling with Multiple Threads". Master. Aalborg University, 2016.

[69] Roger Schlafly. "Methods for Compiling Formulas Stored In An Electronic Spreadsheet System". US5633998. May 27, 1997.

[70] SpreadsheetGear LLC. *SpreadsheetGear*. URL: http://www.spreadsheetgear.com/company/about.aspx (visited on 06/30/2016).

[71] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. "The Design of a Task Parallel Library". In: *SIGPLAN Not.* 44.10 (Oct. 2009), pp. 227–242. ISSN: 0362-1340. DOI: 10.1145/1639949.1640106. URL: http://doi.acm.org/10.1145/1639949.1640106.

[72] Umut A. Acar. "Self-Adjusting Computation". PhD thesis. Carnegie Mellon University, 2005.

[73] Jane Street. *Incremental - Library for incremental computations*. URL: https://opensource.janestreet.com/incremental/ (visited on 01/17/2019).

[74] Holger Stadel Borum, Malthe Ettrup Kirkbro, and Peter Sestoft. *Spreadsheet Patents*. Technical report TR-2018-200. 2018.

[75] Morten Poulsen and Poul Peter Serek. "Optimized Recalculation For Spreadsheets With the Use of Support Graph". Master Thesis. IT University of Copenhagen, 2007.

[76] Peter Sestoft. *A Spreadsheet Core Implementation in C#*. Technical report TR-2006-91. Version 1.0. IT University of Copenhagen, Sept. 2006. 135 pp. URL: https://www.itu.dk/people/sestoft/corecalc/.

[77] Peter Sestoft. "Implementing Function Spreadsheets". In: *Proceedings of the 4th International Workshop on End-user Software Engineering*. WEUSE '08. New York, NY, USA: ACM, 2008, pp. 91–94. ISBN: 978-1-60558-034-0. DOI: 10.1145/1370847.1370867. URL: http://doi.acm.org/10.1145/1370847.1370867.

[78] Peter Sestoft and Jens Zeilund. *Sheet-defined functions: implementation and initial evaluation*. Technical report. Version 1.1. IT University of Copenhagen, Jan. 16, 2013.

[79] ECMA International. *Standard ECMA-335, Common Language Infrastructure (CLI)*. Version 6. June 2012. URL: https://www.ecma-international.org/publications/standards/Ecma-335.htm (visited on 09/05/2017).

[80] Robin Abraham and Martin Erwig. "Type Inference for Spreadsheets". In: *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP '06. New York, NY, USA: ACM, 2006, pp. 73–84. ISBN: 1-59593-388-3. DOI: 10.1145/1140335.1140346. URL: http://doi.acm.org/10.1145/1140335.1140346.

[81] Tie Cheng and Xavier Rival. "Static Analysis of Spreadsheet Applications for Type-Unsafe Operations Detection". In: *European Symposium On Programming (ESOP'15)*. Apr. 2015.

[82] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Revised First Edition. Morgan Kaufmann, 2012. ISBN: 978-0-12-397337-5.

[83] Microsoft. *Task Parallel Library*. URL: https://msdn.microsoft.com/da-dk/library/dd460717.aspx (visited on 08/12/2016).

[84] Oracle. *Class LongAdder*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html (visited on 02/21/2017).

[85] Rico Mariani. *Garbage Collector Basics and Performance Hints*. Microsoft Corporation. Apr. 2003. URL: https://msdn.microsoft.com/en-us/library/ms973837.aspx (visited on 12/12/2018).

[86] Alexander Asp Bock. "A Parallel Spreadsheet Interpreter With Cycle Detection". In: *European Conference on Object-Oriented Programming (ECOOP)*. 2019. Submitted.

[87] Rodrigo Rocha and Bhalchandra D. Thatte. "Distributed cycle detection in large-scale sparse graphs". In: *Proceedings of the Simpósio Brasileiro de Pesquisa Operacional* (Aug. 2015). DOI: 10.13140/RG.2.1.1233.8640.

[88] Trevor Brown, Faith Ellen, and Eric Ruppert. "Pragmatic Primitives for Non-blocking Data Structures". In: *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*. PODC '13. Montréal, Québec, Canada: ACM, 2013, pp. 13–22. ISBN: 978-1-4503-2065-8. DOI: 10.1145/2484239.2484273. URL: http://doi.acm.org/10.1145/2484239.2484273.

[89] Mark Moir. "Practical Implementations of Non-blocking Synchronization Primitives". In: *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '97. Santa Barbara, California, USA: ACM, 1997, pp. 219–228. ISBN: 0-89791-952-1. DOI: 10.1145/259380.259442. URL: http://doi.acm.org/10.1145/259380.259442.

[90] H. Gao and W.H. Hesselink. "A general lock-free algorithm using compare-and-swap". In: *Information and Computation* 205.2 (2007), pp. 225–241. ISSN: 0890-5401. DOI: https://doi.org/10.1016/j.ic.2006.10.003. URL: http://www.sciencedirect.com/science/article/pii/S0890540106001234.

[91] Guy E. Blelloch and John Greiner. "A provable time and space efficient implementation of NESL". In: *ACM SIGPLAN Not.* 31.6 (June 1996), pp. 213–225. ISSN: 0362-1340. DOI: 10.1145/232629.232650. URL: http://doi.acm.org/10.1145/232629.232650.

[92] Marc Fisher and Gregg Rothermel. "The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanisms". In: *Proceedings of the First Workshop on End-user Software Engineering*. WEUSE I. New York, NY, USA: ACM, 2005, pp. 1–5. ISBN: 1-59593-131-7. DOI: 10.1145/1082983.1083242. URL: http://doi.acm.org/10.1145/1082983.1083242.

[93] Guy E. Blelloch. "Programming Parallel Algorithms". In: *Commun. ACM* 39.3 (Mar. 1996), pp. 85–97. ISSN: 0001-0782. DOI: 10.1145/227234.227246. URL: http://doi.acm.org/10.1145/227234.227246.

[94] Robin Abraham and Martin Erwig. "UCheck: A spreadsheet type checker for end users". In: *Journal of Visual Languages & Computing* 18.1 (2007), pp. 71–95. ISSN: 1045-926X. DOI: http://dx.doi.org/10.1016/j.jvlc.2006.06.001. URL: http://www.sciencedirect.com/science/article/pii/S1045926X06000383.

[95] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi. "A type system for statically detecting spreadsheet errors". In: *18th IEEE International Conference on Automated Software Engineering*. Oct. 2003, pp. 174–183. DOI: 10.1109/ASE.2003.1240305.

[96] R. Abraham and M. Erwig. "Header and Unit Inference for Spreadsheets Through Spatial Analyses". In: *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*. Sept. 2004, pp. 165–172. DOI: 10.1109/VLHCC.2004.29.

[97] M. Burnett and M. Erwig. "Visually customizing inference rules about apples and oranges". In: *Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on*. 2002, pp. 140–148. DOI: 10.1109/HCC.2002.1046366.

[98] Chris Chambers and Martin Erwig. "Reasoning about spreadsheets with labels and dimensions". In: *Journal of Visual Languages & Computing* 21.5 (2010), pp. 249–262. ISSN: 1045-926X. DOI: http://dx.doi.org/10.1016/j.jvlc.2010.08.004.

[99] Chris Chambers and Martin Erwig. "Automatic Detection of Dimension Errors in Spreadsheets". In: *J. Vis. Lang. Comput.* 20.4 (Aug. 2009), pp. 269–283. ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2009.04.002. URL: http://dx.doi.org/10.1016/j.jvlc.2009.04.002.

[100] C. Chambers and M. Erwig. "Dimension inference in spreadsheets". In: *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. Sept. 2008, pp. 123–130. DOI: 10.1109/VLHCC.2008.4639072.

[101] Michael J Coblenz, Andrew Jensen Ko, and Brad A Myers. "Using objects of measurement to detect spreadsheet errors". In: *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE. 2005, pp. 314–316.

[102] Martin Erwig and Margaret Burnett. "Adding apples and oranges". In: *Practical Aspects of Declarative Languages*. Springer, 2002, pp. 173–191.

[103] David A. Schmidt. "Trace-Based Abstract Interpretation of Operational Semantics". In: *LISP and Symbolic Computation* 10.3 (May 1998), pp. 237–271. ISSN: 1573-0557. DOI: 10.1023/A:1007734417713. URL: https://doi.org/10.1023/A:1007734417713.

[104] Peter Sestoft. "Replacing function parameters by global variables". In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. ACM. 1989, pp. 39–53.

[105] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert Thomas. *SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual*. Technical report. Version 1.2. Lawrence Livermore National Laboratory, Mar. 1, 1985.

[106] David C. Cann. *SISAL 1.2: A Brief Introduction and Tutorial*. Lawrence Livermore National Laboratory, May 1992.

[107] Project CROAP. *Sisal*. INRIA. URL: http://www-sop.inria.fr/croap/transllprog/subsection3_3_3.html (visited on 11/02/2016).

[108] Stephen Skedzielewski and John Glauert. "IF1 - An Intermediate Form for Applicative Languages". In: *Lawrence Livermore National Laboratory Manual M-170, Livermore, CA* (July 1985).

[109] M. R. Garey and D. S. Johnson. *Computers, Complexity and Intractability: A Guide to the Theory of NP-completeness*. Ed. by Victor Klee. W.H. Freeman and Company, 1979. ISBN: 0-7167-1044-7.

[110] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. 1st. Morgan & Claypool Publishers, 2011. ISBN: 1608455645, 9781608455645.

List of Acronyms

ASP Analytical Spreadsheet Package. 30

CAS compare-and-swap. 49, 50, 52, 57, 59–62, 66, 76, 77, 97, 111, 113, 114, 118, 121, 122, 210

CIL Common Intermediate Language. 42, 134, 150

CPU central processing unit. 25, 26, 33

CSV comma-separated values. 82

DAG directed acyclic graph. 127, 128

DCAS double-length compare-and-swap. 122

DLL dynamic link library. 43, 135

FMD Functional Model Deployment. 28

GPGPU general purpose graphics processing unit. 153, 176

GPU graphics processing unit. 26, 34, 182

GUI graphical user interface. 31

HPC high performance computing. 32

LL load-link. 122

LLX load-link extended. 122

MCAS multi-word compare-and-swap. 122

NaN not a number. 15, 34

P3 Popular Parallel Programming. 3

PODC Principles of Distributed Computing. 2

PU processor unit. 67

SC store-conditional. 122

SCX store-conditional extended. 122

SDF sheet-defined function. 33, 36, 42–44, 48, 51, 69, 87, 90, 91, 94, 128–131, 133–135, 150, 151, 154, 172, 174, 181, 188, 189, 207, 208, 213

SISAL Streams and Iteration in a Single Assignment Language. 12, 23, 25, 43, 155–157

SOA service-oriented architecture. 32

SOS Son of Strike. 85

TA timed automaton. 33

TPL Task Parallel Library. 26, 34, 52, 55, 74, 86, 87, 97, 157, 159, 163, 164, 172, 174, 175, 178

TPU tensor processing unit. 26

TTDA Tagged Token Dataflow Architecture. 25

UDF user-defined function. 28, 32, 33, 42

VBA Visual Basic for Applications. 27, 28, 32, 188, 213

WinDbg Debugging Tools for Windows. 85, 86

WPM Windows Performance Monitor. 81, 82, 94, 114, 120, 187

List of Figures

List of Tables

List of Listings

Appendix A

Source Code

The Funcalc source code is located in a private repository at `https://github.com/popular-parallel-programming/funcalc` and is available upon request.

SheetFill is an F# library used as a scripting language for Funcalc. We used it to generate the synthetic spreadsheets for our benchmarks. Its source code is located in a private repository at `https://github.com/popular-parallel-programming/SheetFill` and is available upon request.

Appendix B

Excel Performance

Although we argued in section 4.3 why a comparison between Funcalc and Excel is not particularly meaningful, we have been asked multiple times for these results by peers and reviewers alike. Therefore, we provide them in this appendix. One useful aspect of a comparison is to gauge the gap in performance between Funcalc, intended for research, and a heavily optimised commercial application.

The benchmarks were measured using custom VBA code[1]. It was not possible to perform the same type of minimal recalculation used in chapters 5 and 7 where we initiate a recalculation using roots that collectively can reach all cells in the spreadsheet via the support graph. The evaluation times reported for Excel are thus for full recalculation only which further diminishes the value of the comparison. Moreover, the LibreOffice Calc spreadsheets only use built-in functions of Excel and no VBA functions and therefore does not showcase the benefit of compiled SDFs.

Figure B.1 compares sequential evaluation of the LibreOffice Calc spreadsheets between Funcalc and Excel. Figure B.2 plots speed-ups on the same set of spreadsheets for Excel. Finally, table B.1 shows the evaluation times in seconds for our thread-based cell interpreter, the static partitioning algorithm with nested cell array parallelism and Excel across all thread configurations. We chose this particular extension of the partitioning algorithm since it had the best overall performance.

[1] `https://docs.microsoft.com/en-us/office/vba/excel/concepts/excel-performance/excel-improving-calcuation-performance`

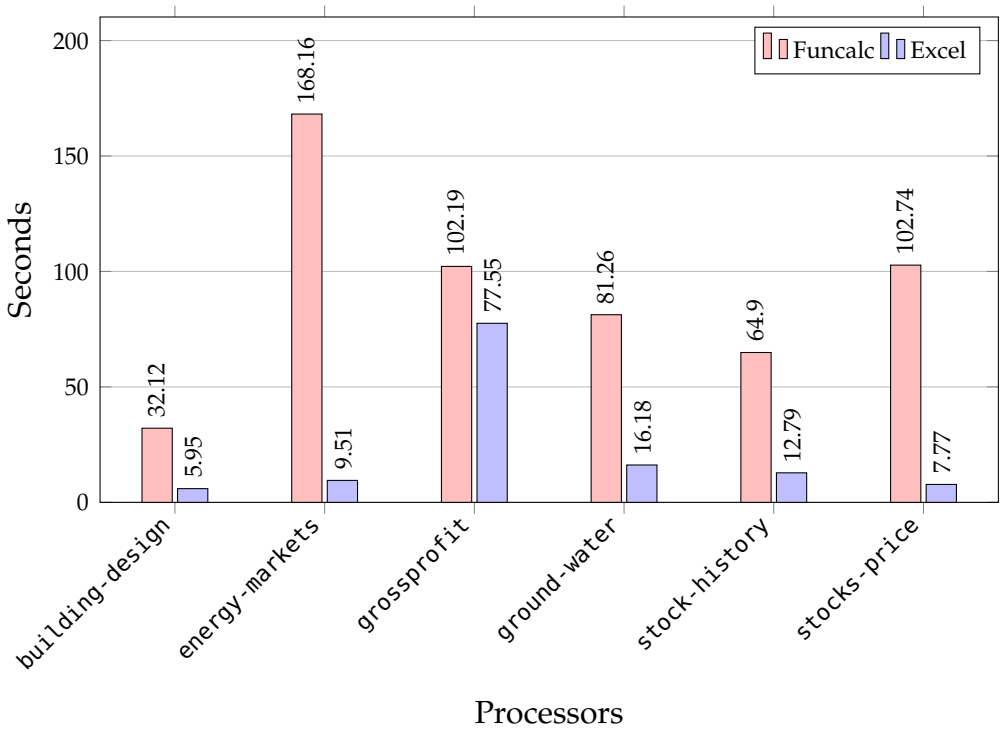


Figure B.1 – Average and standard deviations of sequential performance of 20 runs in Funcalc and Excel on the LibreOffice Calc spreadsheets.

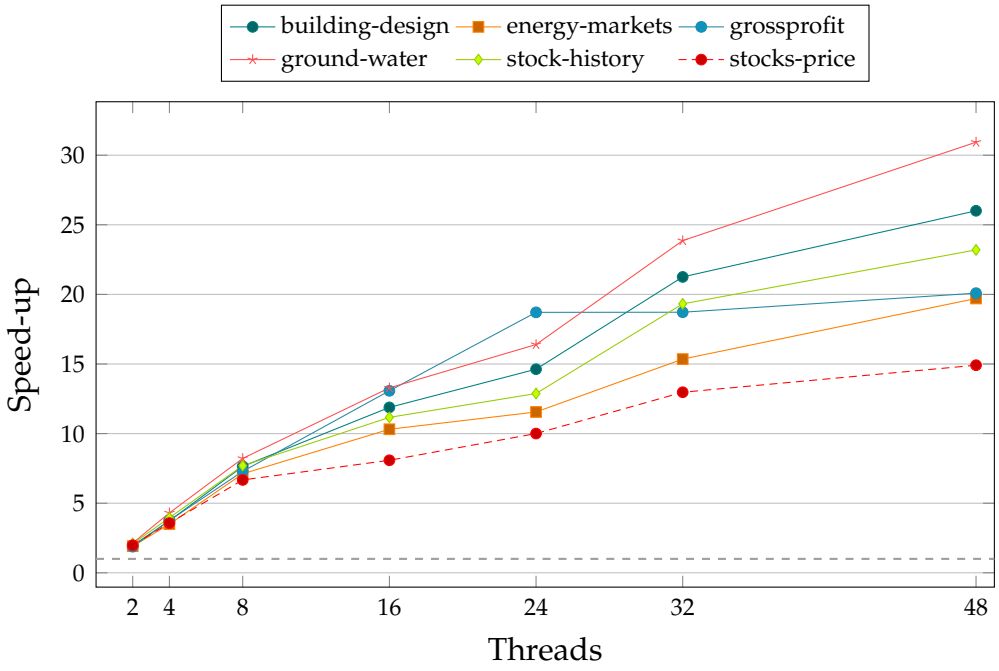


Figure B.2 – Speed-ups on the six LibreOffice Calc spreadsheets for 20 runs in Excel compared to sequential recalculation.

	Number of Threads							
Spreadsheet	**1**	**2**	**4**	**8**	**16**	**24**	**32**	**48**
Thread-Based Parallel Cell Interpreter (chapter 7)								
`building-design`	32.12	29.24	14.38	8.07	4.36	2.95	2.31	**1.65**
`energy-markets`	168.16	499.37	250.93	139.84	75.50	53.45	44.85	**35.83**
`grossprofit`	102.19	253.61	128.23	71.52	39.65	27.75	23.01	**18.42**
`ground-water`	81.26	73.68	37.28	20.41	10.68	7.27	5.42	**3.81**
`stock-history`	64.90	67.01	33.33	17.90	9.49	6.87	5.26	**4.03**
`stocks-price`	102.74	380.00	199.45	110.24	59.98	43.03	35.67	**30.32**
Nested Cell Array Parallelism Extension (section 9.6.1)								
`building-design`	32.12	26.23	13.32	7.33	3.98	2.68	2.16	**1.62**
`energy-markets`	168.16	156.68	95.84	66.67	**53.22**	137.52	89.35	200.28
`grossprofit`	102.19	102.72	53.31	32.46	21.06	17.46	**17.16**	19.95
`ground-water`	81.26	69.97	35.59	19.29	10.41	7.52	5.32	**3.71**
`stock-history`	64.90	58.84	29.94	17.73	10.47	7.64	7.00	**6.17**
`stocks-price`	102.74	130.46	166.70	164.37	**74.44**	132.28	145.48	166.87
Excel								
`building-design`	5.95	3.03	1.59	0.78	0.50	0.41	0.28	**0.23**
`energy-markets`	9.51	4.98	2.72	1.34	0.92	0.82	0.62	**0.48**
`grossprofit`	77.55	41.31	20.62	10.64	5.93	4.14	4.14	**3.86**
`ground-water`	16.18	7.57	3.76	1.97	1.22	0.99	0.68	**0.52**
`stock-history`	12.79	6.12	3.23	1.66	1.15	0.99	0.66	**0.55**
`stocks-price`	7.77	3.92	2.17	1.16	0.96	0.78	0.60	**0.52**

Table B.1 – Absolute running time in seconds for the thread-based cell interpreter of chapter 7 and Excel.